拡張 VLIW プロセッサ GIFT における命令レベル並列処理機構

小松秀昭† 古関聰††
鈴木英俊†† 深澤良彰††

現在，計算機の高速化の手段の一つとして，並列処理の研究が，さまざまな粒度において行われている．我々は，その中で命令レベルでの並列処理に注目しており，優れたアーキテクチャを世に送り出すことを目的としている．命令レベル並列処理アーキテクチャの中で，VLIW はコンパイル時にスケジューリングされた固定命令語を実行するのでハードウェアが簡単になり，高速で高並列度のマシンをつくりやすい．この理由より，VLIW が我々の目的を達するための最適なアーキテクチャであると考えている．しかし，現時点では科学技術計算などの分野を除いて，アーキテクチャが高い並列性を提供しても，それを使いきることは困難である．そこで，我々はプログラムを高度並列化するため，プログラムが本来持っている論理的な並列性を，コンパイラが最大限引き出すことを可能とするための次のアーキテクチャサポートを VLIW に加えた．それらは，① 条件分岐をジャンプなしで実行できる条件実行機構，② 条件分岐を越えた命令の実行をサポートするための先行実行機構，③ メモリアクセス命令の並列実行をサポートするためのメモリシステムである．本論文では，これらのアーキテクチャサポートの詳細と実装，および，これらによる並列性の向上についての評価結果を述べる．

Parallel Executing Mechanisms in the Extended VLIW Processor GIFT

Hideaki Komatsu,† Akira Koseki,†† Eishun Suzuki†† and Yoshiaki Fukazawa††

Instruction parallelism is one of the most promising region in research of parallel processing. In general, clock speed of a VLIW can become faster than Super-scaler, because it has a simple instruction dispatching mechanism. However, programs except for numerical computing cannot use all execution units of VLIW. Therefore, we introduce the following architecture supports that can allow compilers to derive more parallelism from programs. ① A conditional execution mechanism to execute instructions without normal conditional jump instructions. ② A speculative execution mechanism. ③ A dynamic memory disambiguation mechanism to execute memory access instructions simultaneously. In this paper, these mechanisms and their evaluation are described.

1. はじめに

アプリケーションの大規模化にともない，計算機の高速化が求められている．高速化を実現する手段の一つとして，並列処理の研究が，分散処理レベルから命令レベルまで，さまざまな粒度において行われている．その中で我々は，命令レベル並列処理アーキテクチャである VLIW (Very Long Instruction Word) に注目し，研究の対象としている．

VLIW は，これまで，主に科学技術計算を対象として研究されてきたアーキテクチャであり，そのことを背景としてトレーススケジューリング[1]などのコンパイラ技術が開発されてきた．また，VLIW の拡張の試みも数多くなされている[2)~5)]．我々は，VLIW の科学技術計算に対する有効性をそのまま継承し，さらにそれを拡張することによって OS やコンパイラなどすべてのソフトウェアに対して，並列・高速化を図ることのできるアーキテクチャとコンパイラフレームワークを確立することを目的としている．

本稿では，科学技術計算などに特定されないすべてのプログラムを高度並列化するため，プログラムが本来持っている論理的な並列性を，コンパイラが最大限引き出すことを可能とするアーキテクチャサポートについて述べる．

† 日本 IBM (株)東京基礎研究所
Tokyo Research Laboratory, IBM Japan Ltd.
†† 早稲田大学理工学部
School of Science & Engineering, Waseda University

2. 本研究の背景

本章では，VLIW の汎用化に関する問題点を洗いだし，その問題点を解決するための機構を持った拡張 VLIW プロセッサとしての GIFT (Guarded Instruction architecture for Fine-grain Technique) を紹介する．

計算処理は，分岐方向が予測可能かどうか，扱うメモリオブジェクトの参照アドレスが静的に決定できるかどうかで分類することができる．

科学技術計算の分野では，分岐予測がほぼ正しく行われ，また，メモリオブジェクトは配列などで静的にメモリに割り当てられており，それらを参照する場合に，参照アドレスが決定できる計算処理が多い．このタイプの計算処理では，分岐予測に基づいたトレーススケジューリングなどのコンパイル手法を用いることにより，分岐を越えた命令間の並列性の抽出を行うことができる．また，配列を参照するような命令間の並列性は，Bulldog コンパイラで行われているような参照バンク特定化（バンクディスアンビゲーション）手法[1]により，その多くをコンパイラが抽出可能である．したがって，このタイプの計算処理を多く行うプログラムからは，以上の手法により高い並列性を持った VLIW の命令語を生成することが可能であり，科学技術計算を対象とした，ELI-512[6]，Trace[7]などのマシンが世にでている．しかし，この分野の計算処理の最内ループは，配列に対する連続的で定型的な操作であることが多く，これらはベクトルユニットを持ったプロセッサで，より効率のよい処理を行うことができる．プログラムの実行時間の大半が，最内ループの実行で費やされることを考えると，科学技術計算の分野では，ベクトルプロセッサのほうが VLIW よりも一般に優れている．

我々は，VLIW を科学技術用などの専用マシンとして扱うのではなく，汎用高性能マシンとして使用することこそが，その利用価値を最も活かすことのできる方法であると考えている．すなわち，VLIW の汎用化が我々の目的である．これを達成するためには，先のカテゴリすべての計算処理を効率よく処理することが必要である．しかし，分岐方向の予測がたてられないようなプログラム（OS やコンパイラ等）においては，コードスケジューリングによる並列性抽出のスコープは，分岐と分岐の間のわずかな空間に限られてしまう．一般に分岐命令は数命令に 1 回ぐらいの割合で出現するので，得られる並列性は非常に乏しい（2～3 という報告がある[8]）．さらに，分岐方向が予測できる，すなわち，その分岐方向がほぼ一定であるような計算処理では，ブランチターゲットバッファ (BTB) のような機構が有効であるのに対し，そうでないものでは，分岐命令の実行の度にパイプラインの分断が発生し，プログラムの実行効率が著しく落ちてしまう．また，ポインタにより動的に管理されたメモリオブジェクトを使用するプログラム（リスト処理など）では，参照アドレスを静的に決定することができない．したがって，先の並列性抽出手法を用いることは難しく，得られる並列性も乏しい．

以上の分析をまとめると，分岐による並列性と実行効率の低下，および動的メモリアクセスによる並列性の低下が VLIW 汎用化を考えた場合の問題点であると言える．この問題点を解決することが我々の目標である．

具体的な手法を述べる前に，まず，この問題点が解決されたことを仮定した，我々のプロセッサでの実行モデルを述べ，後に，そこへのアプローチを述べる．

ある計算処理の理論的な実行サイクル数の下限は，その計算処理中に存在する最も長いデータフロー（データの生産・消費連鎖）に必要とされるサイクル数で決定される．しかし，実際のプログラムの実行では，条件分岐や動的なメモリアクセスにより，実行に必要なサイクル数は，データフローの長さよりも大幅に延長されてしまっている．我々の拡張 VLIW プロセッサでのプログラムの実行においては，データフローの長さを伸ばすすべての要因を取り去り，実行に必要なサイクルを下限の値に限りなく近くすることが目標である．この実行サイクル数がデータフローの長さに等しくなっているようなプログラムの実行が，我々の目指すモデルである．

そこで，この下限に近付くためには，データの生産・消費連鎖を伸ばす要因を排除する必要がある．この具体的な要因として，以下の項目が挙げられる．

1. 条件分岐の方向が決定されるまで分岐先の命令を実行できない．（制御依存）
2. 条件ジャンプを行うことにより，命令パイプラインが分断される．
3. 条件ジャンプを行うためのサイクルを消費しなければならない．
4. if-then-else 文どうしを並列に実行できない．
5. 不確定な参照アドレスを持つメモリアクセス命

令と他のメモリアクセス命令を同時に実行できない．

6. 変数の使用が終了するまでその値を再定義できない．(逆依存)

我々の目的は，これらの要因を徹底的に排除することであるが，このためのソフトウェアアプローチとして，レジスタリネーミング，トレースに基づいた命令移動，パーコレーションスケジューリングの命令移動[9]などがある．しかし，これらのソフトウェア技法を用いただけでは，十分な要因の排除は行えず，このようなアプローチでは限界があることが否めない．

我々のプログラム最適化アプローチでは，データの生産・消費連鎖を伸ばす要因を排除する際に，ソフトウェアでできるものとそうでないものを区別する．そして，前者に対しては，過去の技法も含めた要因の排除を探求し，後者に対しては，要因を排除するためのハードウェアからのサポートを利用して，実行サイクルの下限に近付ける．このために，どのような部分でハードウェアからのサポートが必要であるかを考察した結果，次のような機構が必要であるという結論に至った．

(1) パイプラインの分断を抑えるため，条件分岐をジャンプなしで実行できる機構
(2) 条件分岐による並列性低下を抑えるため，if-then-else 文全体を一度に実行でき，さらに独立した複数の if-then-else 文を一度に実行するための機構
(3) 同じく条件分岐による並列性低下を抑えるため，条件分岐先の命令をあらかじめ投機的に実行しておくための機構
(4) 動的メモリアクセスによる並列性低下を抑えるため，アクセスアドレスの確定できない複数のロード/ストア命令を一度に実行するための機構

すなわち，これらの機構をコンパイラが利用することによって，上記の要因を十分に取り除くことが可能となる．また，この中の一つでも欠けてしまうと，プログラムの中に取り除くことのできない要因が残ってしまう．一般のプログラムでは，これらの要因は複雑にからみ合っているのが普通であり，この中の機能の一部を実装しただけでは，プログラムを十分に最適化することは難しい．例えば，リストのソートなどを考えた場合，このプログラムの最内ループをアンローリングしたものは，ポインタによるロード/ストアや条件分岐が数多く存在しており，このようなプログラムを効率よく並列化するためには，上記の機能を組み合わせることが必要である．したがって，これらの機構は不可欠であって，全部を組み合わせたアーキテクチャが必要であるというのが我々の主張である．

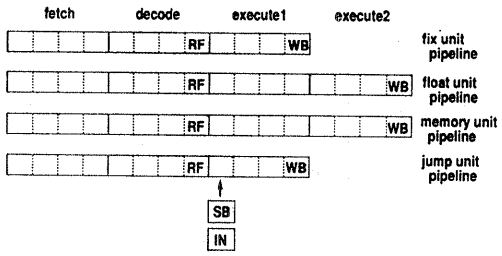
これらの機構を個別に実現した過去のアプローチとしては，次のようなものが挙げられる．

1. Ebcioğlu が提案している，木構造を持った命令語を実行できる VLIW[10]
2. TORCH におけるブースティング[11]
3. Bulldog コンパイラで提案されている，メモリへのアクセスをフロントドアとバックドアに分けて行う VLIW[1]

1は，VLIW の各命令に付けられた命令有効化フィールドと，命令無効化により if-then-else 文を一度に実行する方法であり，これによって条件分岐による並列性の低下をおさえることができる．2は，シャドウハードウェアリソースを用いることによって，条件分岐による並列性の低下をおさえることができる．3は，アクセスアドレスが決定できた場合とできなかった場合で，メモリアクセスのインターフェースを分離する方法であり，これによって，動的なメモリ参照による並列性の低下を軽減することができる．

これらの方法は，それぞれプログラムの実行効率を向上させている．しかし，これらを単独に用いただけでは，一般のプログラムを十分に並列化することは困難である．また，個々の機構もこれから述べる GIFT の機構に比べて，プログラムの並列性抽出に関して不十分な点がある．さらに，個々の機構を組み合わせることによって，相互に機能を補間しながらプログラムの並列性を引き出すことができるということも無視できない．

そこで，我々は，(1)〜(4)の機構をすべて兼ね備えた汎用 VLIW アーキテクチャとしての GIFT を提案する．GIFT は，コンパイラが一般のプログラムを十分に最適化することを可能にする．GIFT が持つべき，これらの機構の具体的な実現として，① 条件実行機構，② 先行実行機構，③ 動的バンク調停を持ったメモリ機構を紹介する．これら各機構の詳細は後続の各章で述べ，既存の機構と比較する．また，GIFT の各機構がどのようにプログラムの並列化に寄与しているか，また，各機構が欠けた場合に，どのような実行効率の低下が起こるかを評価し，GIFT の機構の有効性・不可欠性を確認した．



RF：レジスタファイル読みだし SB：スコアボードチェック
WB：レジスタファイル書き込み IN：割り込み発生チェック

図 1 パイプライン構成
Fig. 1 The pipeline of each execution unit.

3. GIFT のアーキテクチャについて

GIFT は VLIW を拡張したものである．GIFT は機能非均質型 VLIW で，固定小数点演算ユニット，浮動小数点演算ユニット，ジャンプ処理ユニット，メモリ処理ユニットそれぞれを複数個持っている．それぞれのユニットの数は，いろいろなプログラムを調べた結果，4個用意すれば十分であるという考察に基づき，現在，各ユニットを4個ずつ持ったものを GIFT マシンファミリーの最下位機種と考えている．

また，GIFT の基本的パイプライン構成は，**図1**のようになっている．固定小数点ユニットとジャンプユニットのパイプラインは，フェッチステージ，デコードステージ，実行ステージから構成され，浮動小数点ユニットとメモリユニットのパイプラインは，フェッチステージ，デコードステージ，第一実行ステージ，第二実行ステージから構成されている．各命令は，この各ステージをメジャーサイクルとして命令の投入が行われる．さらに，各ステージは四つのマイナーサイクルに分割されていて，このサイクルがデータラッチ間隔の最小のものとなる．

GIFT では，レジスタファイルの読み込みは，デコードステージの最後のマイナーサイクルで行い，レジスタへの書き込みは，実行ステージの最後のマイナーサイクルで行う．また，すべての命令のスコアボード，割り込みのチェックは実行ステージの最初のマイナーサイクルで演算と並行して行われる．各ステージの詳細は，以降の章で随時述べる．

4. GIFT の条件実行機構

4.1 条件実行について

ノイマン型計算機では，ある条件に適合した命令を実行することを，条件分岐命令の結果によって制御してきた．しかし，この方法では，条件文のブロック（then ブロックや else ブロック）内の命令間に並列性があるにもかかわらず，同時に実行することができない．また，条件分岐によってパイプラインが分断されるため，実行速度の低下が起こる．そこで，この問題を条件分岐命令を実行せずに，ある条件に適合した命令のみを実行するハードウェア機構を用いることにより解決する．例えば，**図2**のように，条件文のネストから構成される命令群を実行することを考える．このような一連の命令群をツリーインストラクションと呼び，もし，ツリーインストラクション上の各命令間にデータの依存関係やリソースの衝突がなければ，前述の機構により全体を同時に実行することが可能である．このような実行を行うことにより，条件ブロック間の並列性を活かすことができる．また，条件ジャンプの回数を減らすことができるので，パイプラインの分断の抑制による実行の高速化ができる．GIFT では，このような条件文の並列実行を条件実行と言い，また，そのためのハードウェアサポート機構を条件実行機構と呼ぶ．

GIFT の条件実行機構は，1命令語中の各命令に付加されている実行時に満足していなければならない条件を記述した条件部（ガード）と，条件フラグレジスタ（cf register）と，条件に合わない命令を無効化する仕組みから成っている．条件部は，各条件フラグレジスタと一対一に対応したフィールドで構成されており，この条件部を使って，各命令の実行条件を記述することができる．

図3は，条件フラグレジスタが7個である場合の命令フォーマットである．この図において，条件部の cc_j と条件フラグレジスタ cf_j が対応している．cf_j はプログラム上の比較命令の実行結果が格納されることによって真，偽がセットされる．各命令は，条件フラグレジスタのそれぞれをどのように参照するかを決定するため，各 cf に対応したフィールドに次の三つの

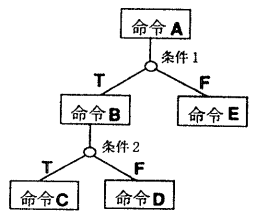
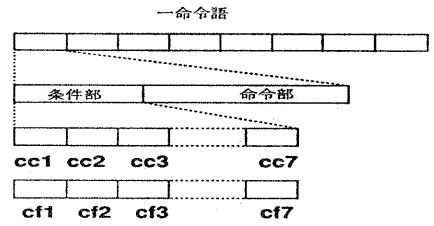


図 2 ツリーインストラクション
Fig. 2 A tree instruction.



cc : condition code cf : condition flag register

図 3 ガード付き命令と条件フラグレジスタ
Fig. 3 A guarded instruction and condition flag registers.

うちのいずれかを記述することができる.

T : 対応する cf が真のときに真

F : 対応する cf が偽のときに真

***** : 対応する cf の真偽にかかわらず真

各命令は，実行時にすべての cc_i と cf_i が比較され，cf_i の状態が完全に cc_i の記述に適合したもののみが実行される．また，その他の命令は無効化される．

例えば，図2を実行するときは，cf 1 に条件 1，cf 2 に条件 2 を比較命令により事前にセットした上で (cf 3〜7 は適当な値を持っている)，

*******A : T******B : TT*****C : TF*****
D : F******E

とすればよい．なお，これからは，表記の簡便化のため，命令語の条件部の中で，***** でしか参照されてない条件フラグレジスタに対応したフィールドを省略する．例えば，先の例は

**A : T* B : TT C : TF D : F* E

となる．また，ガードのない命令は，すべての条件フラグレジスタを ***** で参照しているとする．

ところで，プログラムによっては，複数のツリーインストラクションを同時に実行したい場合がある．

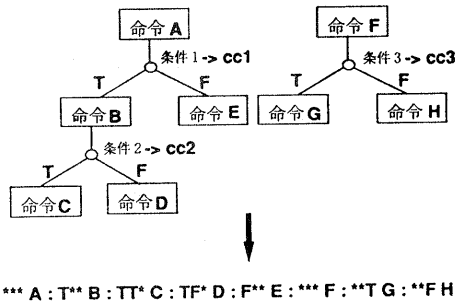


図 4 マルチツリーインストラクションの実行
Fig. 4 Execution of a multi tree instruction.

GIFT では，このような複数のツリーインストラクションをマルチツリーインストラクションと呼び，条件実行によって同時に実行することが可能である（**図 4**）.

4.2 条件実行機構の実装

本節では，前節で述べた条件実行機構のハードウェア実装について述べる．GIFT では，数個の条件フラグレジスタと，ガード部を持った命令を用意している．その上で，実行時に条件と合わなかった命令を無効化するハードウェアを組み入れている．無効化ハードウェアの実装はいろいろ考えられるが，GIFT では，パイプラインピッチを短くするために，処理のクリティカルパスをのばさないような方式を選択している．

図 5 は固定小数点演算ユニット，浮動小数点演算ユニット，ジャンプユニットにおける実装法の概略である．メモリユニットにおける実装は後に第 6 章で述べる．実行条件の判定は実行ステージの立ち上がりからはじまり，各命令の実行と並行して行われる．実行条件に適合しなかった命令に関しては，演算結果をレジスタに書き込むことを禁止したり，パイプラインの途中で実行を取り消すことによって無効化している．

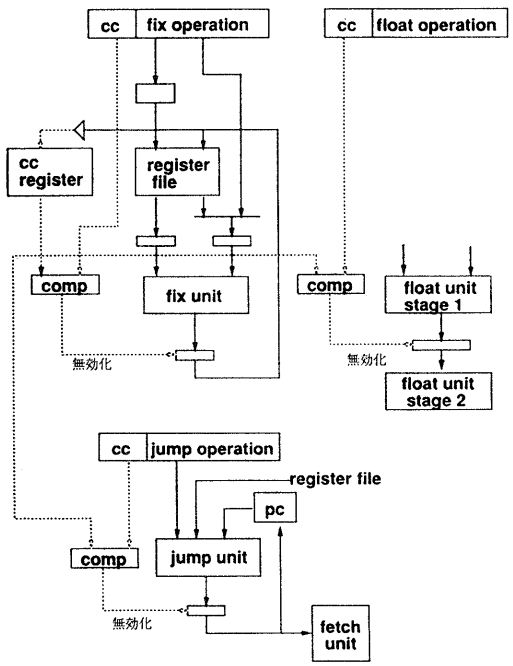


図 5 条件実行機構の実装
Fig. 5 Implementation of conditional execution mechanism.

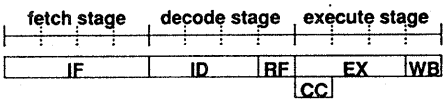


図 6 条件実行機構のタイミング
Fig. 6 Timing of conditional execution mechanism.

この計算のタイミングを図6に示す．実行条件は，デコードステージで決定され，条件フラグレジスタの内容はデコードステージの終わりまでに読みだされる．これらの比較は，演算と並行して実行ステージの第1マイナーサイクルで行われる．

4.3 他研究との比較

ここでは，他のガード付き命令を持ったアーキテクチャと GIFT の条件実行を比較する．

Ebcioğlu は文献 10) において，if-then-else 文を一度に投入するハードウェアを提案している．この方法では，例えば図2のツリーインストラクションを次のように実行する．

1) すべての cc が評価され，ツリーの根から葉までのユニークなパスが選択される．
2) 各命令は，どのパスに乗っているかが記述されており，この情報に従い，各命令の実行，無効化が行われる．
3) すべての「枝先」には，次に実行すべきインストラクションのラベルが記述されているので，選択されたパスの「枝先」のラベルにジャンプする．

GIFT の条件実行との違いは，GIFT では，ある命令がある条件と無関係に実行されるということが陽に記述可能な点である．これを積極的に利用することによって，Ebcioğlu の方法が一つのツリーインストラクションしか実行できないのに対し，GIFT の手法では複数のツリー（マルチツリーインストラクション（図4））を同時に実行することが可能になる．

我々は，予測のできない分岐が頻繁に起こるプログラムを並列化することを目的としている．例えば，ループボディに条件分岐があるようなループをループアンローリングによって最適化しようとした場合，展開されたコードは，if-then-else 文が直列に重なった形になる．このような場合，GIFT ではマルチツリーインストラクションによって，独立した条件文のブロックを一度に実行可能なので，これによりプログラムを飛躍的に並列化できる．もちろん，Ebcioğlu の方式でも，命令を実行される可能性があるすべてのパスにコピーすることによって GIFT の条件実行と同様のことが行える．しかし，この方法ではコードの量がすぐに爆発を起こしてしまい，実用的とは言えない．

また，彼の方法では，一つのツリーインストラクションが実行されると，必ずジャンプが実行される仕様になっている．したがって，分岐方向の一定しないようなツリーインストラクションを実行する場合は，命令パイプラインの分断による実行効率の低下が起こる．GIFT の条件実行機構は，条件分岐によるパイプラインの分断を抑えることもその目的の一つである．GIFT では，命令の無効化を利用して，大抵のツリーインストラクションはジャンプを行わずに連続的に供給される．

5. GIFT の先行実行機構

5.1 先行実行について

パーコレーションスケージューリング[9]などの大域的に命令の移動を行うスケジューリング法では，条件分岐を越えて命令を投機的に実行することが行われる．GIFT では，独自のハードウェア機構によって実現された，投機的な命令の実行を先行実行という（図7）．

先行実行は，データの生産者と消費者の連鎖で作られるデータフローのクリティカルパスを短縮する有効な手段であり，これを用いることによってプログラムの並列性は飛躍的に増大する．しかし，投機的な実行を行った場合，本来なら実行されるべきでない命令が実行されることにより，プログラムの実行効率を低下させたり，プログラムの意味を変えてしまう恐れがある．すなわち，条件分岐を越えて実行された命令が，キャッシュミス，TLB ミス，演算エラーなどを起こし，かつ，分岐が投機に反する方向に起こってしまった場合，本来ならば行われないはずであった命令の復旧処理が，プログラムの実行時間に大きな遅延を与え

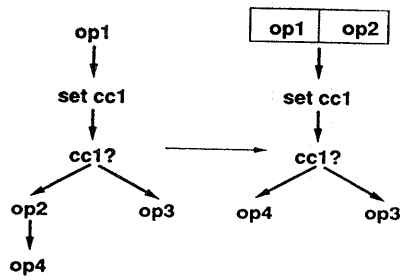


図 7 先行実行
Fig. 7 Speculative execution.

たり，プログラムの意味を変化させることがある．そこで，先行実行を積極的に利用するためには，この問題を解決するためのアーキテクチャサポートが必要である．

この問題を解決するため，GIFT では，投機的実行によりエラーを引き起こす可能性のある命令として通常命令と先行実行命令の二種類を用意している．先行実行命令とは，その命令が前述の事態を起こした場合は，その時点では復旧処理を開始せず，その命令の結果を後の命令が使用する時点で復旧処理を行うということを保証する命令のことである．この先行実行命令を使用することにより，実行されないはずの命令が引き起こす無用な復旧処理は行われず，命令の投機的な実行を積極的に行うことができる．

5.2 先行実行機構の実装

前節で述べたとおり，GIFT では先行実行サポートを，通常命令と先行実行命令という命令セットの二重化と復旧処理の先送りで実現している．復旧処理は，次の二つに分類される．

- TLB ミス，キャッシュミスによるデータ生成のための復旧処理
- 算術演算エラー，アドレス保護例外によるエラー復旧処理（これらのエラーを演算エラーと呼ぶことにする）

まず，前者の処理の先送りの実装のため，各レジスタに TLB ミスビット，キャッシュミスビットの2ビットを付加し，先行実行のロード命令に対し以下のような処理を行う．

1) TLB ミス時

先行実行のロードが TLB ミスを引き起こした場合には，通常の割り込み処理を発生させずに，ロードした内容を書き込むはずであったレジスタに，先行実行命令のロードアドレスを格納し，そのレジスタの持つ TLB ミスビットをオンにする．この後に，ロードした内容を使用するため，他の命令がこのレジスタを読み込んだ場合，TLB ミスビットをハードウェアが検出し，割り込みを発生させ，現在実行されてる命令を無効化した後，復旧ルーチンを実行する．復旧ルーチンでは，割り込みの原因となったアドレスをレジスタから読み込み，これに従って TLB ミスを復旧する．そして有効なロード値をレジスタに格納し，TLB ミスビットをオフにして，無効化された命令から実行を再開する．

先行実行が起こした TLB ミスのアドレスを格納しているレジスタに，読み込みが起こらず，新たな書き込みがなされた場合は，新しい値をレジスタに書き込み，TLB ミスビットをオフにする．

2) キャッシュミス時

先行実行ロードがキャッシュミスを引き起こした場合も TLB ミスの場合と同様に，即時には復旧処理を行わず，ロード値を格納すべきレジスタのキャッシュミスビットをオンにし，キャッシュミスを引き起こしたアドレスを格納する．そして，その後，ロード値を使用するため，レジスタへの読み込みが発生した時点で復旧を行う．新しい値がレジスタに書き込まれた場合は，キャッシュミスビットをオフにする．

演算エラーの処理の実装のためにも，各レジスタに演算エラービットを付加する．演算エラービットは，あるレジスタをディスティネーションとした演算が，その演算処理過程において，演算エラーを発生させたかどうかを示している．先行実行命令が演算エラーを起こした場合，エラー発生と同時に割り込みによる復旧処理ルーチンの実行は行わず，演算結果を保持するはずであったレジスタの演算エラービットをオンにし，レジスタにはエラーコードを格納する．その後，このエラービットが立ったレジスタを他の通常命令が読み込んだ時点で，レジスタが保持していたコードの復旧処理を行う．また，エラービットが立っているレジスタに新しい値が書き込まれたときには，エラービットをオフにする．さらに，エラービットが立ったレジスタを他の先行実行命令が読み込んだ場合，通常命令が読み込んだときのような復旧処理は開始せず，この命令の書き込み先のレジスタの演算エラービットをオンにし，エラーコードをコピーする．

5.3 他研究との比較

文献11)では，分岐を越えた命令の副作用を一時的に格納するためのシャドウハードウェアリソースを用いることによって，ブースティングと呼ばれる投機的実行をサポートしている．この方法では，データの書き込み先がシャドウリソース（シャドウレジスタファイル/シャドウストアバッファ）であることを明示的に記述する．また，各ブランチ命令には，分岐予測の結果を表すフィールドが用意されている．シャドウリソースに格納されたデータは，次のブランチが起こった時点で，通常リソースに書き込まれるか，あるいは無効化される．

GIFT の先行実行との違いは，まず，ブースティングではジャンプが起こった時点で投機命令の結果が決

定されるのに対し，GIFT では先行実行の結果は，後の命令がそれを読み込もうとした時点で決定される点である．この違いによって GIFT では if 文の then ブロックと else ブロックの両方から命令の投機的移動を行えるが，ブースティングではできないという相違点が生じる．分岐の方向は予測不能だということが我々の立場であるので，プログラムに分岐があった場合は，then 方向と else 方向のどちらからも同時に並列性抽出を行えることが必要である．この点で，GIFT の手法がブースティングよりも優れている．

また，ブースティングでは複数の if 文を越えて投機を行おうとすると，その数だけのシャドウリソースを用意する必要がある．一つのレジスタファイルに対し，複数のシャドウファイルを用意するのは，コスト的に効率のよい方法ではない．これに対し，GIFT では，各命令を任意の if 文を越えて投機的移動することができる．例えば，ループボディに条件分岐があるようなループを展開した場合などを考えると，複数の if 文を越えて投機的移動が必要となる場合は多い．

逆に，ブースティングではストア命令を投機移動可能であるが，GIFT ではできないという欠点も存在する．しかし，投機的移動が有効であるのは，データの生産・消費連鎖の起点となっている命令を if 文を越えて移動する場合である．一般に，この連鎖はメモリから値をロードすることによって始まることが多く，データのストアがこのようになっていることはほとんど存在しない．この理由により，我々は現段階ではストア命令の投機的実行は考えていない．

6. 動的バンク調停を持ったメモリ機構

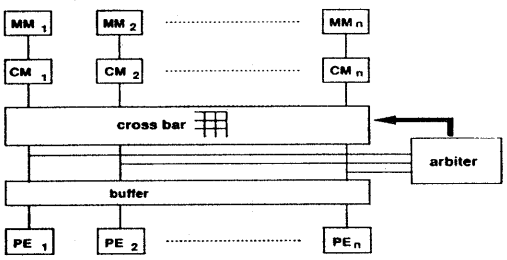
6.1 メモリアクセス命令間の並列性

本節では，Bulldog コンパイラ[1]が扱っているような，ロード/ストア命令を複数持った VLIW と GIFT のメモリ機構を比較する．

メモリアクセスを行う命令間からコンパイラが並列性を抽出するためには，同一アドレスへのアクセスをさけるため，アクセスするアドレスを決定しなければならない．Bulldog コンパイラでは，バンク化されたメモリシステムを持った VLIW をターゲットマシンとして，メモリアクセス命令の参照バンクの特定化を行い，アクセスバンクが決定できた命令を，それぞれのバンクメモリに直接つながったユニット（フロントドア）を使用するようにスケジュールする．また，アクセスバンクが決定できなかった命令は，すべてのバンクメモリに動的にアクセスできるユニット（バックドア）を使用するようにスケジュールされ，結局，最大でフロントドアの数（バンク数）＋バックドアの数（普通は1）の命令が並列実行できる．

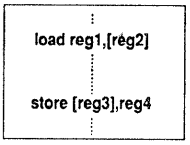
この方法は，静的に宣言された配列を単純なインデックスでアクセスするような場合，命令のアクセスバンクを決定することが可能であり，アクセスバンクの異なった命令群をフロントドアを用いて並列実行できる．しかし，動的なメモリオブジェクトをポインタでアクセスするような場合や，配列を複雑なインデックスでアクセスするような（例えば，配列を配列の要素で参照する）場合は，アクセスバンクの特定化を行うことは非常に困難である．このような場合，コンパイラは命令をバックドアを使用するようにスケジューリングする．しかし，アクセスするアドレスがわからないため，他のフロントドアを使用するメモリアクセス命令と並行実行を行うことはできない．また，バックドアは一つであるため，バックドアを使用するような命令間の並列性も損なってしまうことになる．これらの並列性の損失は非常に大きいと思われる．

GIFT ではこのような問題を解決するため，動的バンク調停機構を持ったメモリシステムを導入している．このメモリシステムは，**図8**に示すようにバンク化されたメモリを持っており，それぞれがキャッシュメモリを持っている．バンクメモリへのアクセスは，クロスバーを通して行われ，すべてのプロセッサエレメントがすべてのバンクにアクセス可能となっている．各バンクメモリへのアクセスは優先順位があり，PE_n から PE_1 の順で優先度が高くなっている．アービタは，各プロセッサからのアクセスを監視しており，同一のバンクにアクセス競合があった場合は，優先順位の高いものだけがバンクメモリにアクセスす

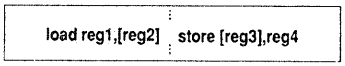


MM: Main Memory
CM: Cache Memory
PE: Processor Element

図 8 GIFT のキャッシュメモリシステム
Fig. 8 Cache memory system of GIFT.



(a) スケジュール前の命令
(a) Instructions before schedule.



(b) スケジュール後の命令
(b) Instructions after schedule.

図 9 スケジューリングの比較
Fig. 9 A comparison of scheduling.

るようにクロスバーをセットする．競合に負けたものは，アクセス要求はバッファにためられ，次のサイクルで再びアクセスを行う．

このことを踏まえて，コンパイラは，参照バンクを特定することのできなかった命令を含む複数のメモリアクセス命令があった場合，これらを，本来実行される順番（スカラープロセッサで実行される順番）に合わせて同一命令語内の優先順位の高いプロセッサに投入されるフィールドに割り付ける．例えば，**図 9**（a）のような命令があった場合，load 命令を PE_1，store 命令を PE_2 に投入されるフィールドに割り付ける（図 9（b））．これによって，実行時に reg 2 と reg 3 の示すアドレスのバンクが異なる場合にこれらを並列に実行することができる．たとえ reg 2 と reg 3 のアドレスが同一であった場合でも，それらのアクセスバンクは同じなので，前述のメモリシステムにより，load 命令が先にバンクメモリにアクセスし，その後 store 命令がアクセスするので，プログラムの意味は保証される．

従来のアーキテクチャをターゲットとしたコンパイラでは，reg 2 と reg 3 の非同一性を保証できないかぎり同一命令語に割り付けることはできなかった．このような，参照アドレスの決定できないロード/ストア命令はリスト処理などのプログラムのループボディを展開すると数多く発生する．GIFT ではそのような場合でも命令群から並列性をとりだし，高速化を図ることが可能である．

6.2 メモリユニットにおける条件実行の実装

本節ではメモリユニットにおける条件実行の実装について述べる．4 章において，条件実行による命令の無効化の実装を述べた．それは，ガードと条件フラグレジスタが一致しなかった命令について，演算結果の書き込みを禁じたり，パイプラインの途中で実行を取り消したりするものであった．メモリユニットにおいては，この命令の無効化を，競合調停の不参加とクロスバーへのデータ転送禁止によって行う．

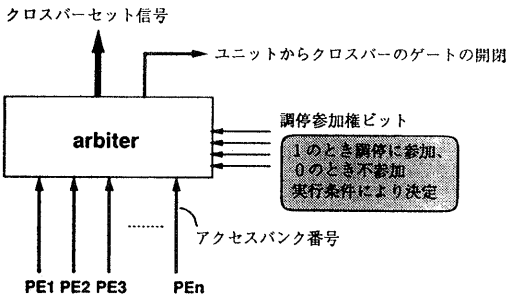


図 10 調停への参加
Fig. 10 Participation in arbitration.

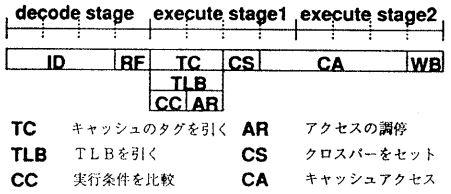


図 11 メモリユニットのタイミング
Fig. 11 Timing of memory access unit.

GIFT のメモリユニットにおいては，前章で説明したとおり，メモリアクセスの各ユニットは，それぞれが自由なバンクをアクセスしており，その際の同一バンクのアクセス競合の調停は図 8 のアービタが行っている．このとき，命令の実行条件が適合しなかったユニットについては，**図 10** に示す通り，アービタの調停参加権ビットをオフにする．アービタは，調停参加権ビットがオフであるユニットが，メモリへのアクセスを行わないように各ユニットを調停し，さらに，調停参加権ビットがオフであるユニットとクロスバーとのゲートを閉ざして，命令の無効化を行う．

7. 評　　価

この章では，先行実行，条件実行など，GIFT のアーキテクチャサポートについてその有効性の定量的な評価を行う．評価は，ベンチマーク（リバモアカーネル），その他のプログラムについて，各プログラムの最内ループをアンロールしたものを，**表 1** に示す各アーキテクチャに対して最適化を施し，各アーキテクチャのインストラクションシミュレータで実行したときの平均実行速度を比較した．ただし，スカラープロ

表 1 対象アーキテクチャ
Table 1 Object architectures.

スカラープロセッサ
LIW (flx : 2, float : 2, jump : 1, memory : 2)
VLIW (flx : 4, float : 4, jump : 4, memory : 4)
VLIW＋条件実行
VLIW＋先行実行
VLIW＋動的バンク調停
VLIW＋条件実行＋先行実行＋動的バンク調停 (GIFT)

表 2 カーネル1, 3, 12の実行速度比較
Table 2 Comparison of execution speed for kernels 1, 3, 12.

アーキテクチャ	カーネル1	カーネル3	カーネル12
スカラー	1.000	1.000	1.000
LIW	4.739	4.000	4.525
VLIW	5.464	6.211	6.410
GIFT	5.464	6.211	6.410

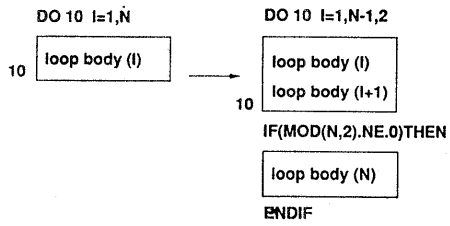


図 12 ループアンローリング
Fig. 12 Loop unrolling.

表 3 一般のプログラムの実行速度比較
Table 3 Comparison of execution speed for ordinary programs.

アーキテクチャ	member ('a', List)	max (List)	sort (List)
スカラー	1.000	1.000	1.000
LIW	1.217	1.139	1.198
VLIW	1.217	1.139	1.198
VLIW＋条	1.938	2.488	1.667
VLIW＋先	2.817	2.045	1.393
VLIW＋動	1.217	1.139	1.490
GIFT	2.959	3.731	2.625

セッサの場合の速度を1とした.

表2はカーネル1, 3, 12を実行したときの比較結果である. これらのカーネルは do all 型の for 文でのループであり, **図12**のようにループを展開することにより, 展開したイタレーション内の命令を並列実行することができる. したがって, 演算処理を行うユニットを複数にすることで実行速度は向上している.

しかし, このようなループの展開が有効なのは, for 文による定数回のループである場合や, ループボディの中に if 文がない場合などに限られる. **表3**は, 一般のプログラムの中から, ループボディに if 文があるもの, while 文による不定回数ループであるものなどを選んで, 実行速度を比較したものである. ここに示したようなプログラムでは, 演算処理ユニット数を単に増やしただけでは, 実行速度の上昇はみられない (LIW と VLIW の比較). それに比べ, GIFT はこのようなプログラムからも並列性を抽出し, 実行速度向上を図ることができる (LIW と GIFT の比較).

以下では, それぞれのプログラムについて, GIFT のアーキテクチャサポートが実行速度上昇に, どのように作用しているかを考察する.

member 関数は, リストのポインタをたどりながら, ある要素をサーチする関数であり, ポインタが NULL であるかの判定, 要素の読みだし, 要素の判定, 次のポインタの読みだしを繰り返すループ構造となっている. このループボディを展開して最適化を行った場合, 要素を読みだすためのロード命令を, ループの繰り返しを判定する if 文を越えて実行することができないため, 十分な並列化を行うことができない. したがって, 演算処理ユニットを増やした効果は現れていない. GIFT では, 先行実行により, ロード命令を if 文を越えて実行することができる. このように, ロード命令を投機的に実行することで, 全体の処理のクリティカルパスを大幅に縮小することができる. この効果によりスカラープロセッサの3.0倍の速度向上があった.

max 関数は, リストのポインタをたどりながら, すべての要素を比較し, 最大のものをサーチする関数であり, ポインタが NULL であるかの判定, 要素の読みだし, 今までの最大値との比較, 次のポインタの読みだしを繰り返すループ構造となっている. このループ構造では, ループの中に条件分岐のブロックが存在し, これによるパイプラインの乱れ, 並列性の阻害が大きく, 演算処理ユニットを増やした効果は現れていない. GIFT では, 条件実行による並列性の向上と, 条件ジャンプ命令の減少により, スカラープロセッサの3.7倍の速度向上があった.

sort (List) 手続きはリストのポインタをたどりながら昇順 (あるいは降順) に値を入れ換えるものであり, アドレスの確定できないストア命令や要素の比較などの if 文が存在するので既存のアーキテクチャでこれを十分に並列化できない. ここでは, 具体的な

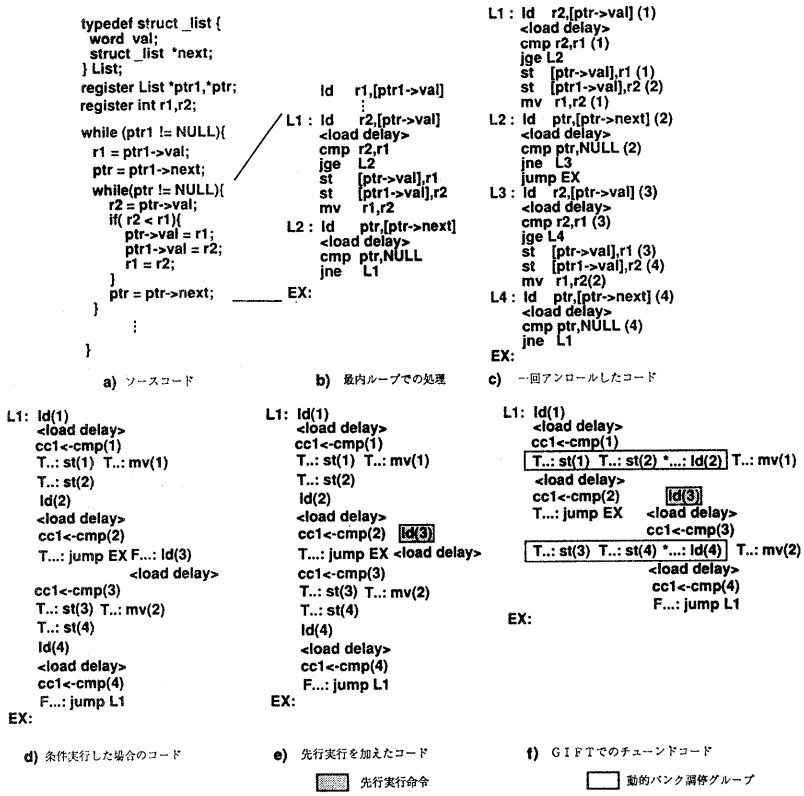


図 13 ソート手続き
Fig. 13 Sort procedure.
(a) Sorce code, (b) Process in inner loop, (c) Code unrolled once, (d) Code using guarded instructions, (e) Code added speculative instructions, (f) Tuned code in GIFT.

GIFT の中間コードを例にとりながら，GIFT のアーキテクチャサポートの有用性を定性的にも明らかにし，さらに，個々の機能の不可欠性を示すことを試みる．

図 13(a)は選択ソートのソースプログラム，図 13(b)は最内ループの中間コード（遅延分岐なし，ディスティネーションは左のオペランド）である．これを一回展開したものが図 13(c)であるが，これを最適化することを考える．まず，このままで並列実行可能なのは，mv(1) と st(2)，mv(2) と st(4) ぐらいである．また，要素どうしの比較結果はばらつきがあるので，パイプラインの分断が頻繁に発生して効率的な実行が行えない．このコードのスカラプロセッサにおける実行時間を見積もってみると，EX にジャンプする確率を 0，L1・L3 にジャンプする確率を 1，L2・L4 にジャンプする確率を 0.5，ジャンプが発火した場合のペナルティを 0 サイクル，ジャンプが発火しなかった場合のペナルティを 2 サイクルとして，

実行時間≃(16(jump L2, jump L4 taken)
+21(jump L2 taken)+21(jump L4 taken)
+26)/4=21

サイクルとなる．このコードを，命令レベルの並列プロセッサで実行することを考えると，mv と st が並列実行できるので，実行時間の見積もりは 20 サイクルとなり，1.05 倍の速度向上が得られる．しかし，既存の VLIW のように単に処理ユニットを増やし，ハードウェアの並列性を上げても，並列実行可能な命令数は増えないので，これ以上は速度の向上を望むことはできない．以下では，GIFT のアーキテクチャサポートが処理速度の向上にどのように寄与しているかを説明する．

まず，GIFT の条件実行を用いると，条件分岐を，ガードつき命令を用いてジャンプなしで実行することができる．したがって，パイプラインの分断による実

行速度低下は取り除かれ，同じコードの実行時間の見積もりは17サイクルとなる（図13(d)）．さらに，先行実行を用いると，ld(3)のような条件分岐の先にあるロード命令を投機的に移動させて，処理を最適化できる．この効果によって，さらに実行時間の見積もりは減って，16サイクルとなる．最後に動的バンク調停を加えると，ストア命令とロード命令を同一命令語においてプロセッサに投入することができる．ところで，各構造体が，4ワード境界でメモリに割り当てられていると仮定すると，ptr->val と ptr->next，ptr 1->val と ptr->next は同一バンクをアクセスすることはない．結局，ld(2)，ld(4)は他の命令とバンク競合を起こさないので，この命令は投入されたサイクルでそのまま実行される（ストア命令どうしがバンク競合した場合でも），したがって，このコードの実行速度は12サイクルに短縮される．これにソフトウェアパイプラインなどの最適化を加えた最終的な実行速度比較は表3のとおりで，スカラープロセッサの2.6倍の速度向上が得られた．また，図13(e)に到達するためには，GIFT の個々の機能の一つでも欠けてはいけない．したがって，GIFT のアーキテクチャサポートの不可欠性の一例が示されたといえる．

これらの結果より，命令レベルの並列処理においては，単に演算処理ユニットを増やすだけでは実行速度は上がらず，GIFT のようなプログラム最適化のためのハードウェアを組み合わせたアプローチが必要であることと，また，それが実際に有効であることが言える．

8. おわりに

本稿では，並列度向上のためのアーキテクチャについて述べ，処理速度の向上に関して評価を行った．

今後，GIFT アーキテクチャの拡張として，スーパーパイプライン化の研究を進めており，この技術を用いた，時間並列処理によるスケーラビリティの実装，ベクトル・スカラー両演算性能向上を考えている．最終的なパイプラインのスペックは，図1で示したマイナーサイクルで，周波数 200～500 M [Hz] での命令投入を目指している．

参考文献

1) Ellis, J. R.: *Bulldog: A Compiler for VLIW Architectures*, The MIT Press (1985).
2) Cohn, R., Gross, T., Lam, M. and Tseng, P. S.: Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor, *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2-14 (1989).
3) Dehnert, J. C., Hsu, P. Y.-T. and Bratt, J. P: Overlapped Loop Support in the Cydra 5, *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26-38 (1989).
4) Wolfe, A. and Shen, J. P.: A Variable Instruction Stream Extension to the VLIW Architecture, *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2-14 (1991).
5) 有田隆也，曽和将容：命令語動的再構成型 VLIW プロセッサアーキテクチャ，電子情報通信学会論文誌，Vol. J 76-D-I, No. 4, pp. 184-186 (1993).
6) Fisher, J. A.: Very Long Instruction Word Architectures and the ELI-512, *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 140-150 (1983).
7) Colwell, R. P., Nix, R. P., O'Donnell, J. J., Papworth, D. B. and Rodman, P. K.: A VLIW Architecture for a Trace Scheduling Compiler, *IEEE Trans. Comput.*, Vol. C-37, No. 8, pp. 967-979 (1988).
8) Nicolau, A. and Fisher, J. A.: A Measuring the Parallelism Available for Very Long Instruction Word Architectures, *IEEE Trans. Comput.*, Vol. C-33, No. 11, pp. 968-976 (1984).
9) Aiken, A. and Nicolau, A.: A Development Environment for Horizontal Microcode, *IEEE Trans. Softw. Eng.*, Vol. 14, No. 5, pp. 584-594 (1988).
10) Ebcioğlu, K.: Some Design Ideas for a VLIW Architecture for Sequential Natured Software, *Parallel Processing* (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing), Edited by Cosnard, M. et al., North Holland (1988).
11) Smith, M. D., Lam, M. S. and Horowitz, M. A.: Boosting beyond Static Scheduling in a Superscalar Processor, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 344-354 (1987).

(平成5年2月8日受付)
(平成5年9月8日採録)

小松 秀昭（正会員）

昭和35年生．昭和60年早稲田大学理工学研究科電気工学専攻修士課程修了．同年日本IBM(株)入社，以来東京基礎研究所（TRL）において，Prologコンパイラ，HPEコンパイラの研究に従事．早稲田大学において命令レベル並列のアーキテクチャ，コンパイラの研究に従事．ACM会員，情報処理学会アーキテクチャ研究会（ARC）連絡委員．

古関 聰

1969年生．1992年早稲田大学理工学部電気工学科卒業．現在，同大学院理工学研究科電気工学専攻博士前期課程在学中．計算機アーキテクチャ，並列処理の研究に従事．

鈴木 英俊

1966年生．1990年早稲田大学理工学部電気工学科卒業．1992年同大学院理工学研究科電気工学専攻修士課程修了．同年日本電信電話(株)に入社．現在，デジタル交換機の保守・運用に従事．

深澤 良彰（正会員）

昭和51年早稲田大学理工学部電気工学科卒業．昭和58年同大学院博士課程中退．同年相模工業大学工学部情報工学科専任講師．昭和62年早稲田大学理工学部助教授．平成4年同教授．工学博士．ソフトウェア工学，コンピュータアーキテクチャなどの研究に従事．電子情報通信学会，ソフトウェア科学会，IEEE，ACM各会員．