順序付きマルチスレッド実行モデルの提案とその評価

本村真人[†] 井上俊明[†]
鳥居淳[††] 小長谷明彦[††]

マルチスレッド実行モデルおよびマルチスレッドアーキテクチャは，従来，超並列計算機の基本技術のひとつとして位置付けられてきた．本稿では，従来とは視点を変え，マルチスレッド技術によるスレッドレベル並列処理を将来のマイクロプロセッサ高性能化のキー技術として捉え，共有メモリ型の小規模並列計算機（1チップ化を想定）上の基本マルチスレッド技術の検討を行った．このような問題設定下では，従来のマルチスレッド技術において並列性抽出能力と引き替えに許容されていた実行時間上/アーキテクチャ上のオーバヘッドが致命的な欠点となる．この問題を解決するために，本稿は順序付きマルチスレッド実行モデル（Ordered Multithreaded-Execution Model）を提案する．本実行モデルはマルチスレッドコード中に仮想制御フロー（Virtual Control Flow）を導入し，仮想制御フローに基づく逐次的な実行順序により並列スレッド間の実行スケジューリングをハードウェア制御することを特徴としている．初期的な評価の結果，小規模並列計算機システムにおいて，本実行モデルにより効率の良いスレッドレベル並列処理を実現できることが明らかとなった．

Ordered Multithreaded-Execution Model: The Proposal and Evaluation

MASATO MOTOMURA, TOSHIAKI INOUE, SUNAO TORII and AKIHIKO KONAGAYA

Traditionally, multithreaded execution models and architectures have been studied as effective techniques for massively parallel processors. This paper approaches multithreading from a different perspective: we have explored multithreading as key techniques for utilizing thread-level parallelism in future high performance microprocessors. In this microprocessor setting, various run-time and architectural overheads associated with conventional multithreading techniques become intorelable because the amount of parallelism exposed is orders of magnitude smaller. In response to this problem, the paper proposes *ordered multithreaded-execution model*. Under this novel execution model, thread scheduling is hardware-controled in accordance with a sequential execution order along a *virtual control flow* which is newly introduced into a multithreaded code. Preliminary evaluation results show that the proposed execution model has the potential to effectively exploit thread-level parallelism in a shared-memory multiprocessors system.

1. はじめに

近年のハイエンドマイクロプロセッサの急速な性能向上は，クロック速度の向上とともに命令レベル並列処理技術により支えられてきた．しかしながら，よく知られているように[1),2)]，プログラムに内在する命令レベル並列性は本質的に限られており，このため対効果ハードウェア資源投入比は今や次第に減少する傾向にある．このような事実を踏まえ，我々はマイクロプロセッサ高性能化技術としてスレッドレベル並列性の有効利用技術が将来的に重要になると考えている．すなわち，少数の比較的独立した処理ユニット（Processing Unit: PU）を1チップ上に統合し，これら複数のPUで独立した複数のコントロールフローすなわちスレッドを並列に実行する形態が，スーパースカラ/VLIW等とあわせて，将来のハイエンドマイクロプロセッサのひとつの形態となりうると考えている（同様な見解は文献2)においても述べられている）．

スレッドレベル並列処理技術をこのように「将来のマイクロプロセッサの性能向上のキー技術」として位置付けるためには，最低限以下のような基本的要請が

† 日本電気株式会社マイクロエレクトロニクス研究所
Microelectronics Research Laboratories, NEC Corporation
†† 日本電気株式会社C&C研究所
C&C research Laboratories, NEC Corporation

満たされていなければならないと我々は考える．

コードのスケーラビリティ 同一のコードが任意台数のPUで構成されるマイクロプロセッサ上で支障なく実行されねばならない．

性能のスケーラビリティ 上記コードの1PU上の実行効率が逐次コードの実行効率と同程度であり，かつPUの数に応じた性能向上が実現されねばならない☆．

これらの要請の重要性は，PUを実行ユニットEUに置き換えてみることにより明らかになる．すなわち，スーパースカラプロセッサはこれらの要請をすべてクリアしており，ゆえに「扱いやすい」高性能化技術として急速に受け入れられたということができる．

スレッドレベル並列処理においてコードスケーラビリティを実現することは，プログラムに存在する並列スレッドの数が物理的に存在するPUの台数を上回ることをつねに想定しなければならないことを意味する．これはスレッド間の同期動作のミス時の処理方法に大きな影響を与える．なぜならば，そのような状況下では，あるスレッドで同期ミスが発生したときにそのスレッドがブロックしてしまうと，一般にはデッドロックが発生してしまうためである．このためコードスケーラビリティを実現するためには，不可避的に同期ミス時のスレッド切替え機構をサポートする必要がある．この機構はソフトウェアで実現することも可能であるが，1PU上での高速処理の要請を同時に実現するためには，高速でかつ軽いハードウェア機構として実現される必要がある．

このような同期ミスによるスレッド切替えは，従来，マルチスレッド技術がその特徴としてきたものである(EM-4[3]，*T[4]，Alewife[5]など)．しかしながら，これら従来のマルチスレッド技術は，上に述べたマイクロプロセッサの高性能化技術としての位置付けには本質的に不都合な点を含んでいる．それは，マルチスレッド化にともなう実行時のオーバヘッドが大きいか，あるいは実行時のオーバヘッドを下げるために過度のプロセッサ状態の増大を招いているという点である．これらの欠点は，従来のマルチスレッド技術が超並列計算機上での実行を前提として，プログラム中に存在するすべてのスレッドレベル並列性を有効利用するように最適化されている点に起因している．

本稿は，これらの欠点を解消し，メモリを共有した小規模並列計算機において上記の基本的要請を満たしたスレッドレベル並列処理を実現する順序付きマルチスレッド実行モデル（Ordered Multithreaded-Execution Model）を提案する．本実行モデルは，マルチスレッド化されたコードに仮想制御フローと呼ぶ仮想的な逐次実行パスを導入し，この仮想制御フローに沿った実行順序を基準として1つのPU上におけるスレッドスケジューリングをハードウェア制御することを特徴としている．本実行モデルにより，スレッドスケジューリングやメモリ管理をランタイムシステムなしに簡易に実現し，同一のコードを任意台数のPU上で効率良く実行することが可能となる．

以下，2章では従来のマルチスレッド技術について考察するとともに仮想制御フローの概念について述べる．次に3章で順序付きマルチスレッド実行モデルについて説明し，4章で本実行モデルにおけるアーキテクチャサポートについて説明する．5章では，試作したパイプラインシミュレータ[6]と簡易コンパイラを用いて行った評価の結果を述べその分析を行う．6章では評価結果を踏まえて提案技術の問題点等について議論し，最後に7章でまとめと今後の展望を述べる．

2. マルチスレッド実行モデルの解析

図1に本稿で取り扱うマルチスレッド実行モデルの基本的な枠組を示した．スレッドの粒度は手続きレベルとし，文献3)，4)のようなこれより細粒度のマイクロスレッドはサポートしない．したがって，各スレッドには駆動（Activation）フレーム[4]が1対1に付随することになる．また，フォークされ並列に起動された手続きだけでなく，逐次的に呼び出された手続きもスレッドと呼ぶ．複数の並列スレッド間の同期動作は，図に示されたように共有変数を介して行われる．さらに，任意のスレッドは必ず終了動作により明示的にその実行を終える．

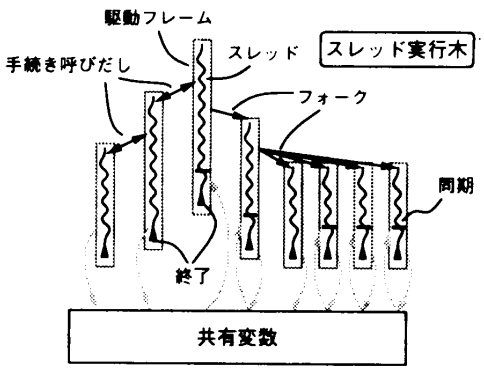

図1 基本マルチスレッド実行モデル
Fig. 1 Basic multithreaded execution model.

☆ ここで性能のスケーラビリティとは，スーパースカラの場合と同様に，ある程度の台数まで性能が向上するということを意味しており，超並列計算機の場合によく評価指標とされるリニアなスピードアップを意味するものではない．

フォークされた後実際に並列起動される前までのスレッドの状態を実行可能状態と呼ぶ．前章で述べたように，本実行モデルでは同期ミスに遭遇したスレッドは他のスレッドに切り替えられる必要があるが，この際は実行可能状態のスレッドのいずれかを選択して切り替える．またこのように同期ミスを起こして切り替えられたスレッドを実行待ちのスレッドと呼ぶ．実行待ちのスレッドは必ず同一のPU上で再実行される．さらに手続き呼び出しを行うことにより処理を中断されたスレッドを休眠中のスレッドと呼ぶ．実行中，実行待ち，休眠中の各スレッドに付随する駆動フレームはメモリ中で同時に生存している．このため，これらの状態にあるスレッドをまとめて生存中のスレッドと呼ぶ．以下に見るように，1つのPU上で同時に生存中のスレッドの相対的な実行順序をどのように制御するかが，マルチスレッド実行モデルを特徴付ける大きなポイントとなる．

2.1 並列処理と制御フロー

図2は制御フローの視点から命令レベル並列処理をまとめたものである．1本の制御フローに沿った逐次的な実行と，制御フローには関係なくデータ駆動により命令の並列実行を行う古典的なデータフロー実行モデルがスペクトラムの対極に位置する．データフロー実行モデルはきわめて高い命令レベル並列性抽出能力を持つものの，よく知られているように，命令の実行スケジューリング上大きなオーバヘッドを有する点がその欠点である．一方スーパースカラ実行モデルは，制御フローの定める逐次的なセマンティクスを保ちながら命令レベル並列性を動的に利用することを特徴とする，かなり広い範囲にわたる実行モデルである．この中でも逐次実行順序を遵守しながら複数の命令を並列に実行する順序実行（In-Order Execution）よりも，逐次実行順序を逸脱した非順序実行（Out-of-Order Execution）の方が，より多くの命令レベル並列性を抽出することが可能となる．しかしながら，その代償として，逐次実行順序から離れれば離れるほど，アーキテクチャ上より複雑な処理を要求され，クロック速度の低下を招く危険性が生じる．ここで，複雑な処理が要求されるのは主に命令のスケジューリングとレジスタの管理廻りである．これらに関して，必要な資源の待ち合わせを行うデータフロー的なハードウェア制御機構を導入しなければならない[7]．

次に同様な視点からスレッドレベル並列処理について整理してみる．従来のマルチスレッド実行モデルにおいては，必要なデータの揃った（マイクロ）スレッドから実行が開始されるという点において，スレッドの実行スケジューリングはデータフロー的な機構により行われている．ここでは命令レベル並列処理における1本の制御フローに対応するような，すべてのスレッドの実行順序をあらかじめ指定するものは存在しない．このため，従来のマルチスレッド実行モデルは，命令レベル並列処理におけるデータフロー実行モデルと同様に，主にスレッドのスケジューリングとメモリ管理の点において，不可避的に実行時のオーバヘッドが大きくなってしまうという問題を有している（3.1節参照）．これらの問題は，簡単にいえば，データフロー的なスレッドスケジューリングの実現の難しさと，このようなスケジューリングにともないスレッドの終了順序と開始順序の間に固定的な相関関係が存在しないことに起因している．

本稿の最も基本的なアイデアは，マルチスレッド実行モデルに仮想的な制御フローを導入し，この制御フローを用いてスレッドの実行スケジューリングを制御する点にある．ここで仮想制御フローとは，それに沿った逐次的な実行がつねに可能であるように形成されたコード全体の逐次実行パスである．

図2と同様に，仮想制御フローをベースとした視点から，スレッドレベル並列処理における実行モデル間の相対関係を**図3**にまとめた．仮想制御フローに沿って順にスレッドを処理する逐次実行モデルと，仮想制御フローとは無関係な従来のマルチスレッド実行モデルはスペクトラムの対極に位置する．我々は，両者の中間に位置し，仮想制御フローの定める逐次的なセマンティクスを保ちながらスレッドレベル並列性を動的に利用することを特徴とする新しいマルチスレッド実行モデルを提案する．この実行モデルを順序付きマル

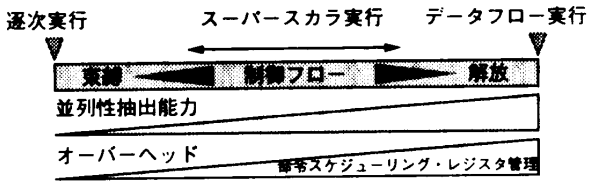

図2 命令レベル並列処理
Fig. 2 Instruction-level parallel processing.

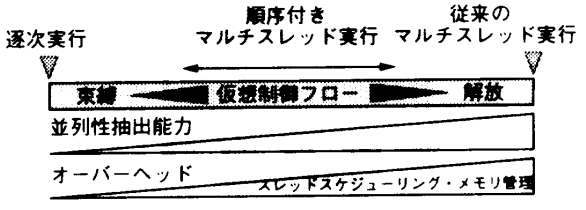

図3 スレッドレベル並列処理
Fig. 3 Thread-level parallel processing.

チスレッド実行モデルと呼ぶ．

図2と図3の比較から明らかなように，順序付きマルチスレッド実行モデルは，命令レベル並列処理におけるスーパースカラ実行モデルと同じ位置付けをスレッドレベル並列処理において占めることを狙っている．すなわち，最大のスレッドレベル並列性抽出能力を狙うのではなく，オーバヘッドが少なく，扱いやすいスレッドレベル並列処理を実現することがターゲットである．3章で見るように，実際に順序付きマルチスレッド実行モデルによりスレッドスケジューリングおよびメモリ管理にともなう従来モデルのオーバヘッドを大幅に削減することができる．一方，並列性抽出能力の低下は，我々の対象とするシステムが超並列計算機ではなく少数のPUから構成されたマイクロプロセッサであることを考えると，欠点とはならない．

2.2 仮想制御フロー

仮想制御フローは，すべてのフォーク動作を手続き呼びだしに置き換えることにより得られる．この置き換えは，手続き呼びだしをフォーク動作に置き換えて逐次コードの並列化を行う場合とちょうど反対の手順であり，その意味において仮想制御フローはマルチスレッドコードの自然な逐次実行順序を与えるということができる．しかしながら，一般のマルチスレッドコードにおいては，仮想制御フローに沿った逐次実行は同期ミスによりブロックされてしまう可能性がある．そこで，我々は任意のマルチスレッドコードに対して，仮想制御フローに沿った逐次実行がつねに同期ミスなしで実行可能であることを要請する．またこのようにして構築されたコードを順序付きマルチスレッドコード（Ordered Multithreaded-Code）と呼ぶ．

図4 に順序付きマルチスレッドコードの例を示す．図では仮想制御フローに従ったスレッドの終了順序により各スレッドが番号付けされており，この番号を仮想スレッド番号と呼ぶ．つねに仮想スレッド番号が小さい方から大きい方へ向かって同期が行われており，これにより仮想制御フローに沿った逐次実行が可能であることが保証されている．

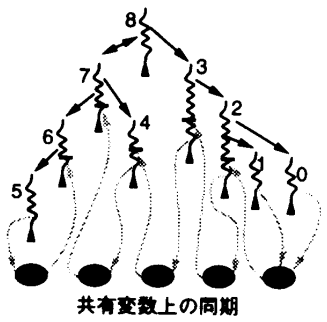

図4 順序付きマルチスレッドコード
Fig. 4 Ordered multithread-code.

3. 順序付きマルチスレッド実行モデル

順序付きマルチスレッド実行モデルでは，「あるPU上で同時に生存中の複数のスレッドの中で実行中のスレッドがつねに最も小さな仮想スレッド番号を有すること」を条件としてスレッドの実行スケジューリングを制御する．この条件は以下の2つの原則により発行可能なスレッドを定め，これら発行可能なスレッドのみを実行スケジューリングの対象とすることで満たすことができる．

(1) あるスレッドが同期ミスを起こしてスレッド切替を行う場合は，そのスレッドよりも小さい仮想スレッド番号を持つ実行可能なスレッドのみが発行可能である．

(2) ある並列スレッドが実行を終了した場合は，そのPU上で実行待ち状態にあるスレッドの中で最も仮想スレッド番号の小さいものと，それよりも小さい仮想スレッド番号を持つ実行可能なスレッドのみが発行可能である．

これらの条件に見合う発行可能なスレッドが存在しない場合は，PUはそれぞれ同期のスピンウェイトを行うかあるいはアイドル状態になる．

図5 に2つの原則に基づく発行可能スレッドの例を示す．図5(a)において，あるPU上でスレッド8が休眠中，スレッド7と6が実行待ち（同期ミス），スレッド4が実行中であるとする．ここでスレッド4が同期ミスを起こしたとすると，原則(1)に従い，スレッド0から3までが発行可能となり，これらのみが実行スケジューリングの対象となる．次に図5(b)において，同じ状況下でスレッド4が実行を終了したとする．この場合，原則(2)に従い，まずこのPU上で実行待ち状態にあるスレッド6と7の中で仮想スレッド番号の小さなもの，すなわちスレッド6が発行可能となる．さらにこれよりも仮想スレッド番号の小さな

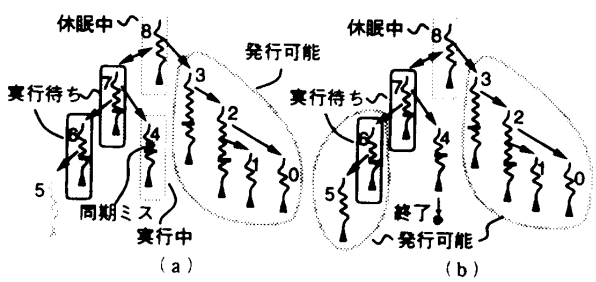

図5 順序付きマルチスレッド実行モデル
Fig. 5 Ordered multithreaded-execution model.

スレッド5とスレッド0から3も発行可能である．これらのスレッドが実行スケジューリングの対象となる．

3.1 順序付き実行の利点

以上のような順序付きマルチスレッド実行の第1の利点は，スレッドスケジューリングを簡略化できる点である．従来のマルチスレッド実行モデルにおいてスレッドスケジューリングのコストが大きかった理由は，同期と実行スケジューリングが結合されており，かつ同期ポイントのネームスペースすべてをプロセッサ状態としてカバーすることが不可能な点にあった．すなわち，同期ポイントを格納するハードウェアバッファを設け，同期成功時のスレッドスケジューリングをアーキテクチャ的にサポートしようとしても，一般に必ず同期バッファのオーバフローが発生し，一部の，あるいは大半の同期ポイントはメモリ中に退避される．このような状況において低コストで同期-スレッドスケジュール機構を実現するのは大変に困難である．このような問題を避けるために，文献4)などでは，1つのスレッドを同期ポイントで複数のマイクロスレッドに分離し，同期が成功した場合に後続のマイクロスレッドをフォークするようにしている．これによりアーキテクチャとして同期-スケジュール機構をサポートする必要はなくなるが，同期成功時の実行時のオーバヘッドが許容し難いほど大きくなってしまう．

これに対し，本実行モデルではスレッドの実行スケジューリングと同期が切り離されており，同期が成功したかどうかにかかわらず，発行可能なスレッドの実行を開始する（したがって，実行を再開した実行待ちスレッドの同期が再度ミスする可能性がある）．このように両者を直交化することにより，スレッドスケジューリングと同期のアーキテクチャサポートを分離し，双方を効率良く実現することが可能となる．

順序付きマルチスレッド実行のもう1つの大きな利点は，駆動フレームの管理が大幅に簡単化されることである．その理由は，本モデルの下では，任意のPU上でスレッドの実行がつねにLIFO順序で行われる点にある．すなわち，あるスレッドが実行を開始した後にその生存期間中に同一のPU上で実行される任意のスレッドは，必ず前者のスレッドよりも小さな仮想スレッド番号を有しており，また必ず先にその実行を終了する．ここで2.2節で述べた順序付きマルチスレッドコードの同期動作方向に関する要請がこのようなLIFO順序のスレッド実行を可能としている点に注意されたい．なぜならば，これにより，あるPU上で同時に生存中のスレッドに関して，つねに後に実行を開始したスレッドから先に実行を開始したスレッドの方向へのみ同期動作が行われるようになるからである．

従来のマルチスレッド実行モデルの下では，このような固定的なスレッド開始-終了順序関係が保証されないため，ヒープライクなデータ構造を用いてオーバヘッドが大きい駆動フレーム管理を行う必要があり，さらにこの管理のためにランタイムシステムの介入を必要としていた．これに対し，本実行モデルではLIFO順序が保証されるため，通常の逐次実行モデルにおける場合と同様にスタックを用いた駆動フレームの管理が可能となる．つまり，スレッドのコード中にスタック管理をインライン展開し，簡便にかつ低コストで駆動フレームの管理を行うことができる．

4. アーキテクチャサポート

本稿の対象とするマイクロプロセッサのアーキテクチャモデル（4PU版）を**図6**に示す．マイクロプロセッサはメモリを共有する複数のPUから構成され，それぞれのPUは命令発行ユニット（IIU），レジスタファイル（RF），命令およびデータキャッシュメモリ（CM）を備えている．あるPUでは同時には1つのスレッドのみが実行中である（つまり多重ハードウェアコンテクストはサポートしない）．本アーキテクチャモデルで特徴的な点は，すべてのPUに共有され，順序付きマルチスレッド実行モデルに従ってスレッドの実行スケジューリングを一元的に管理するスレッド発行ユニット（TIU）の存在である．あるPUがスレッドをフォークすると，実行可能なスレッドのスレッド記述子（TD）がTIUに送出され格納される．また，実行中のスレッドが同期ミスに遭遇した場合，実行待ち状態になったスレッドのTDがTIUへ送出される．TIUはこれら2種類のTDを保持し，あるPU上で実行中のスレッドが同期ミスにより実行待ちになった際かあ

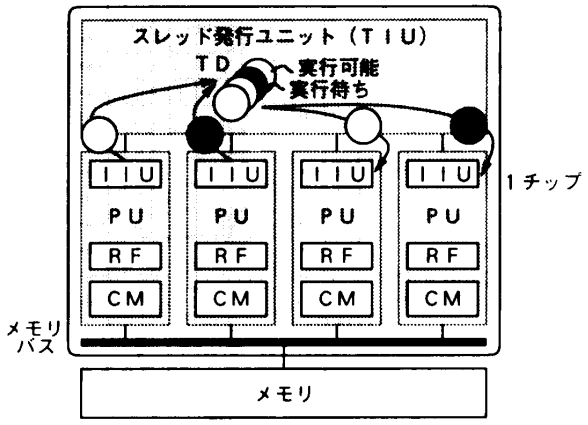

図6 マイクロプロセッサのアーキテクチャモデル
Fig. 6 Architecture model of a target microprocessor.

るいは実行中のスレッドが終了した際に，3章の2つの原則に従って発行可能なTDを定め，そのうちのいずれかのTDをそのPUへ発行する．

4.1 スレッド発行ユニット

2章で順序付きマルチスレッド実行モデルとスーパースカラ実行モデルを対比したように，スーパースカラプロセッサのIIUと対比することでTIUの位置付けを明確化することができる．すなわち，IIUが命令の発行可能性を判断して命令を複数のEUへ発行するように，TIUはスレッドの発行可能性を判断してスレッドを複数のPUへ発行する．この際IIUが物理的な制御フローに基づき，命令間のデータ/制御依存性を正確に把握したうえで各命令の発行可能性を判断できるのに対し，TIUはスレッド間のデータ依存性に関してこのような正確な情報を手にすることができない．そこでTIUは，順序付きマルチスレッド実行モデルの指示に従い仮想スレッド番号に基づいてスレッドの実行を制御する．これは任意のスレッド間で仮想制御フローに沿ったデータ依存性がつねに存在すると仮定し，安全なスレッドスケジューリングを行っていることに相当する．

図7にTIUのブロック図を示す（4PU版）．スレッドバッファ（TB）は任意数のTBスロットから構成され，それぞれのTBスロットは実行可能か実行待ちのTDを保持する．実行待ちのTDにはPU番号が付随する．またすべてのTDには実行フラグ（後述）が付随する．TIUのアーキテクチャ上のキーポイントは，(1) TB内でどのようにスレッド間の順序関係を管理するか，(2) TDの発行をどのようにスケジュールするか，(3) TBのオーバフローをどのように処理するか，の3点である．それぞれの点について以下説明する．

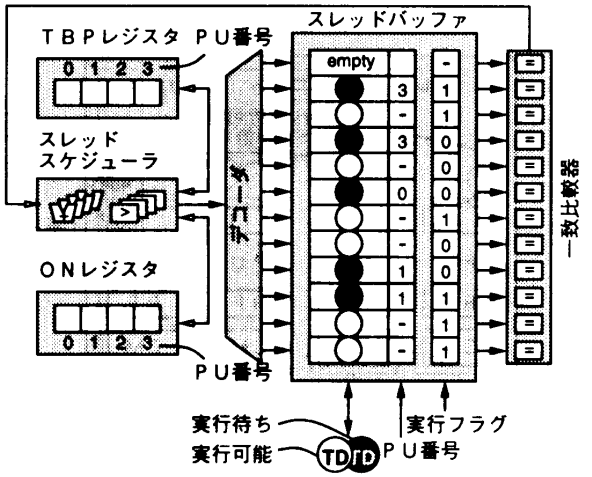

図7 スレッド発行ユニット
Fig. 7 Thread issue unit.

4.1.1 スレッド間順序の管理

TDを発行するにあたって発行可能かどうかを判断するためには，仮想制御フローに基づいてTD間および実行中のスレッド間の順序関係を把握しておく必要がある．TIUは，仮想スレッド番号を直接用いるのではなく，TD間の相対的な順序関係を用いてTB内でTDを仮想スレッド番号順に整列させることで，これを実現している．TDの整列化は，各PUにスレッドバッファポインタ（TBP）を与え，TBPの指定する位置にそのPUから送出されたTDを挿入する操作と，TBへTDを挿入した際かTBからTDを発行した際に，適切なTBスロットを指すようにTBPを付け換える操作とで行われる．

図8に示した例を用いてこれを簡単に説明する．同図において，実行待ちのTDにはPU番号が示されている．また番号付けされた正方形は各PUのTBPを表している．TBの下の方が仮想スレッド番号が小さいように整列されるものとする．まず図8(a)においてPU0がスレッドをフォークすると実行可能なTDがTB内のTBP0で指定された位置に挿入されるとともにTBP0，3，2がそれぞれ1つ上のTBスロットに付け換えられる．次に図8(c)においてPU0で実行中のスレッドが同期ミスに遭遇した場合，TBP0の指すTBスロットより下に位置しかつ実行可能なTDが発行可能である．それらのTDのうちの1つを選んで発行するとともに，実行待ちのTDをTBP0が指していたTBスロットの1つ下の位置へ挿入する．TBP0は，TDが発行されたTBスロットに付け換えられる．

このような動作を図7のスレッドスケジューラや一致比較器等を用いて実現することにより，TB内でTDを仮想スレッド番号順に整列して保持することが可能となる．なお，順序番号（ON）レジスタは複数

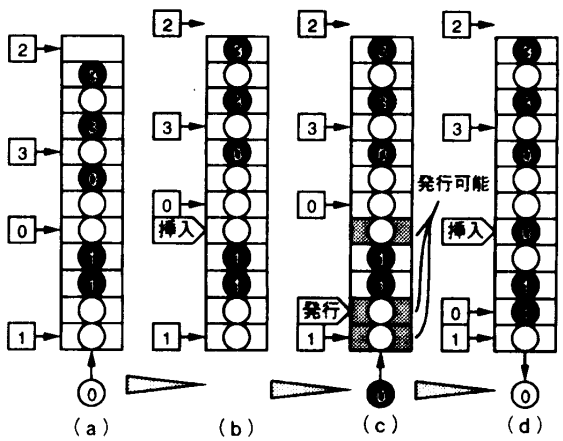

図8 スレッドバッファ管理
Fig. 8 Thread buffer management.

のTBPが同じTBスロットを指す場合に，それらの相対的な順序関係を把握するために用いられる．

4.1.2 スレッドスケジューリング

順序付きマルチスレッド実行モデルでは，発行可能なTDのうち実際にどれを発行するかを自由に決定することができる．たとえば図8(c)においては，3つの発行可能なTDのうちいずれを発行してもよい．したがって様々なスレッドスケジューリング手法のバリエーションが考えられ，それらの手法によりスレッドレベル並列処理の実行効率が大きく影響されることが予想される．簡単なハードウェア機構で実現可能なことがこれらのバリエーションの条件である．

本稿では，以下のようなスケジューリング手法を考案し，これに関して評価を行った．

(1) 1つのスレッドからフォークされた姉妹関係のスレッドの中では，最も先にフォークされたものから，すなわち最も仮想スレッド番号の小さいものから実行スケジュールする．

(2) 世代の違うスレッドの中では最も世代の古いスレッドから，すなわち最も仮想スレッド番号の大きいものから実行スケジュールする．

並列ループに代表される姉妹関係の並列スレッド間では，前のイテレーションから後のイテレーションに向かって同期が行われることが多いため，このように先にフォークされたものから順に実行を行った方が同期ミスが少なく，実行効率が良いはずである．一方，再帰呼び出しに代表される複数の世代にまたがる並列スレッド間では，このように各PUが発行可能なスレッドの中で最も古い世代のスレッドを実行した方が，それぞれのPUがスレッド実行木の中で別々の部分木を実行することができるため，自然に負荷分散を実現できると期待される．

このスケジューリング手法は，図7において，1) フォーク時には姉妹関係にあるTDの中で最も仮想スレッド番号の小さいものにのみ実行フラグを立て，2) TDを発行する際にあたっては発行可能なTDの中で最も上に位置する実行フラグが立てられたTDを選択して発行し，3) 同期ミスによる実行待ち時には実行フラグを立ててTDを書き戻す，という一連の簡単なハードウェア動作により実現される．3) の動作により同一世代の中で複数の実行待ちスレッドが実行可能フラグを立てられることになる．しかしながら，1)，2) の動作により任意のPU上で実行待ちができるスレッドは同一世代の中でただ1つだけに限られるため，提案スケジューリング手法どおりのスレッド発行が可能となる．

4.1.3 並列スレッドの動的逐次化

3.1節で説明したように，従来のマルチスレッド実行モデルでは，同期バッファのオーバフローがスレッドの実行スケジューリング上の大きな問題となっていた．順序付きマルチスレッド実行モデルにおいても，同期をスケジューリングから切り離したとはいえ，TBのオーバフローを適切に処理しない限り似たような問題を抱え込むことになる．

我々はこの問題を回避するために，TBがフルになった場合は新しいスレッドのフォークを拒絶し，代わりにフォーク元のPUで逐次的な手続き呼び出しとして実行する手法を提案する．TBがフルかどうかはTBPの値により簡単に判別可能である（図8参照）．順序付きマルチスレッドコードにおいては，同期の方向に対する制約から，このような動的なスレッドフォークの手続き呼び出しへの置換がつねに可能であることに注意されたい．このような手法を動的スレッド逐次化（Dynamic Thread Streamlining）と呼ぶ．

動的スレッド逐次化により，同時に存在するスレッドレベル並列性の最大量はTBの大きさにより規定されることになる．言い換えると，プロセッサ状態としてTIUに保持されたTDだけを対象として，スレッドスケジューリングを行えばよいということである．これにより，スレッドスケジューリングのアーキテクチャサポートが簡単化されるだけでなく，過剰なスレッドレベル並列性により実行効率が低下してしまう問題を防ぐことも可能となる．

なお，このように動的に並列性を制限する場合，一般にはデッドロックの危険性があるが，本手法を用いた順序付きマルチスレッド実行モデルについてはデッドロックセーフであることを証明することができる．キーポイントとなるのは実行可能と実行待ちの2種類のTDがまとめてTBで管理されており，これらの総数によりスレッドのフォークが可能かどうかが判断されている点である．

4.2 PUアーキテクチャ

PUのアーキテクチャは，RISCマイクロプロセッサを下敷にし，主に以下のような点で順序付きマルチスレッド実行モデルを効率化するためのアーキテクチャサポートを導入することで構成される．

フォーク フォーク命令を命令セットに追加し，この命令によりTDの生成と送出を行う．動的スレッド逐次化をサポートするために，この命令はフォークが可能かどうかを判断し，不可能な場合は手続き呼び出しを開始する必要がある．

同期 同期はすべて専用の同期命令を用いてメモリ空

表1 ベンチマークプログラムの一覧
Table 1 Benchmark programs.

ベンチマークプログラム	静的特性		動的特性			
	データセット	コード長	実行時間	フォーク	手続き	同期
クイックソート	10,000 数列	96 命令	2.79 M	4,288	6,298	4,288
FFT	1,024 点	388 命令	1.69 M	12,670	3	12,675
TSP	4,000 都市	409 命令	4.65 M	19,158	19,143	6,818
N-クィーン	10 クィーン	212 命令	14.51 M	34,814	726	22,765
ウェーブフロント	256×256 行列	115 命令	2.08 M	256	2	65,281

間上で行われる．同期失敗時のスレッド切替をサポートするために，この命令は同期ミス時にそれをIIUに知らせる必要がある．これに応じてIIUはTDをTIUへリクエストし，新しいスレッドの実行を開始する．

終了 スレッド終了命令を命令セットに追加し，この命令によりスレッドの終了を明示的にIIUへ通知する．これに応じてIIUはTDをTIUへリクエストし，新しいスレッドの実行を開始するかあるいは実行待ちのスレッドの実行を再開する．

スレッド切替え 以上の動作におけるスレッドの切替を高速化するために，コンテキストの退避/回復を高速に行う機構を用意する必要がある．

我々は，例としてR3000をベースに選び，以上のようなアーキテクチャサポートを実現したPUアーキテクチャを設計した．その基本的な構成は文献6)で述べられたものと同一であるので，ここでは説明を割愛する．

5. シミュレータによる評価

順序付きマルチスレッド実行モデルとそのアーキテクチャサポートの定量的評価を行うために簡易コンパイラとパイプラインシミュレータ[6]を試作した．簡易コンパイラはGCCをベースにしており，Cのソースプログラムにスレッドフォークとスレッド間同期を指定するインライン関数を人手挿入することでマルチスレッドコードを生成することができる．パイプラインシミュレータは図6に示したマイクロプロセッサの動作をサイクルレベルで正確にシミュレーションすることができる．シミュレータの入力パラメータはPU台数，キャッシュサイズ，TBサイズなどである．なお，現在のところ，キャッシュプロトコルとしては単純なライトスルーしか実装されていない．

5.1 ベンチマークプログラム

このような評価ツールを用いて，**表1**に示した5つのよく知られたベンチマークプログラムに対して評価を行った．それぞれのCプログラムは順序付きマルチスレッドコードの定義に従って人手で並列化されている．同一のコードが，ランタイムシステムの介入なしに，そのまま任意のPU台数構成で実行可能である．

ここではそれぞれのプログラムにおいてマルチスレッド実行上の課題となる側面について簡単に紹介する．クィックソートは再帰コールの並列化の例題として選ばれた．このプログラムではスレッド実行木は2進木として構成されるが，2つの子スレッド間の負荷が均一でない点が特徴である．高速フーリエ変換（FFT）は3重ループ構造であり，内側のループの上限は外側のループにより定められる．このプログラムはループ並列化の例題として選ばれた．巡回セールスマン問題（TSP）とN-クィーンでは，スレッドレベル並列性がふんだんにある場合に，本実行モデルがうまく並列性の制御を行えるかどうかを確認することができる．どちらのプログラムともに再帰コールとループの双方の並列化を行っている．なお使用したTSPプログラムは文献8)に紹介された高速近似解法に基づいてコーディングしたものである．最後にウェーブフロントは2重ループの単純な構造を有しているが，並列スレッド（外側のループを並列化）間で頻繁な同期動作を必要とする点で特異であり，本実行モデルがこのような多大な同期動作とそれにともなうスレッド切替えをうまく取り扱うことができるかどうかを調べることができる☆．

それぞれのプログラムは表1に示されたデータセットを用いて実行された．表1の動的特性は1PU上での実行における総実行サイクル数，総フォーク回数，総手続き呼び出し回数，および総同期回数を示したものである．

5.2 評価結果

表2はこれらのプログラムの性能向上率をプログラムごとにまとめたものである．この表は2，4，6，8台のPU構成において，1PUあたりのTBスロット数が1，4，8，12個の5つのTB構成について，それぞれ1PU構成と比べた性能向上率を測定した結果で

☆ このプログラムはデータ並列計算モデルでは並列性の抽出が難しい問題としてよく知られている．

表2 性能向上率の一覧
Table 2 Performance improvements for the benchmarks.

クイックソート				
TBスロット数 (PUあたり)	PU数			
	2	4	6	8
1	1.74	2.53	3.07	3.92
4	1.79	2.65	3.64	4.26
8	1.90	3.17	3.70	4.33
12	1.90	3.16	3.97	4.36
FFT				
TBスロット数 (PUあたり)	PU数			
	2	4	6	8
1	1.71	3.00	3.74	3.92
4	1.85	3.20	3.92	4.06
8	1.86	3.27	4.00	4.10
12	1.88	3.29	4.04	4.11
TSP				
TBスロット数 (PUあたり)	PU数			
	2	4	6	8
1	1.82	3.05	3.81	4.25
4	1.82	3.05	3.79	4.20
8	1.81	3.03	3.78	4.16
12	1.79	3.01	3.74	4.12
N-クィーン				
TBスロット数 (PUあたり)	PU数			
	2	4	6	8
1	1.97	3.88	5.54	7.20
4	1.98	3.91	5.68	7.34
8	1.98	3.91	5.73	7.31
12	1.97	3.91	5.74	7.38
ウエーブフロント				
TBスロット数 (PUあたり)	PU数			
	2	4	6	8
1	1.96	3.84	5.61	7.28
4	1.96	3.85	5.65	7.36
8	1.96	3.86	5.65	7.36
12	1.96	3.86	5.65	7.39

ある．1PU上では逐次実行が行われるため，実行サイクル数がTBスロットの数に依存しないことに注意されたい．キャッシュは命令/データキャッシュともにダイレクトマップ/8KBで，ラインサイズは32Bである．メモリライト（リード）遅延は，メモリバスを入れて2（4）サイクルを仮定している．

4PU構成までは，すべてのプログラムについて4台で3倍以上のかなり良い性能向上率が得られている．クィックソート，FFT，TSPに関しては，それ以上のPU台数で性能向上が次第に飽和する傾向にある．一方，N-クィーンとウェーブフロントは，8PUまで高い性能向上率を示しながらほぼリニアに性能が向上している．また，TSPの場合を除き，TBスロット数が多い方がより高い性能を得ることができるが，その性能差はクィックソートの場合を除きあまり大きくないことが分かる．以下では，性能の飽和したプログラムについて詳しい解析を試みる．

クイックソート-TSP

図9にこれらのプログラムの実行プロファイルを示した．この実行プロファイルは，4PU-32TBスロット構成において，プログラムの実行中のPU使用率（実行中-同期ミスによる実行待ち-スレッドなし）とTBの占有率（実行可能TD-実行待ちTD-TDなし）に関して，その動的な振舞いの遷移を示したものである．

PU使用率のプロファイルから，クイックソートでは実行を始めた直後に，TSPでは実行を終了する直前に，それぞれ比較的長い期間，少数のPUしか使用されていないことが分かる．これらは，それぞれ数列を2つに分割するポイントを探す期間と，ローカルな近似的最短パスをつなげてグローバルな近似的最短パスを形成する期間に対応している．これら以外の期間においては，4つのPUはほぼいつも有効に使用されている．このように，これら2つのプログラムにおける性能の飽和は，コード実行中のスレッドレベル並列性が存在しないかあるいは少ない期間がその要因となっていることが理解される．

一方，クィックソートのTB占有率に関しては，占有率が激しく変動するとともに，実行可能よりも実行待ちのTDの数の方が上回っていることが分かる．このプログラムのフォークと同期の数は表1に示したように等しいので，これは実行待ちのTDの方がTBでの滞在時間が平均的に長いことを意味する．よりミクロに見ると，これは同じ同期命令で何度もミスを繰り返している場合があるからだと理解される．一方，TSPでは上記の1PU実行期間を除きTBはつねに100%近く占有されている．TSPの方がクィックソートよりもTBの占有率が高いにもかかわらず，TBのスロット数を減らすと，前述のようにTSPの性能はむしろ上がり，逆にクィックソートの性能が下がることに注意されたい．すなわち「TBがフルになった際に動的にスレッドの逐次化を行うことが性能の低下を招くとは限らない」ことが分かる．クィックソートのTBスロット数依存性は，TBの多くの部分を実行待ちのTDが占めてしまうことにより，発行可能なTDの数が相対的に少なくなってしまっているためだと考えられる．

FFT

表3は，1PUあたり8TBスロットの構成において，PU台数に応じた各プログラムのメモリバス占有率の変化を示したものである．この表から，FFTにおいてメモリバス占有率が非常に高く，これが性能を飽和させる要因になっていることが分かる．このような振舞いの最も基本的な原因はFFTのデータ参照局所

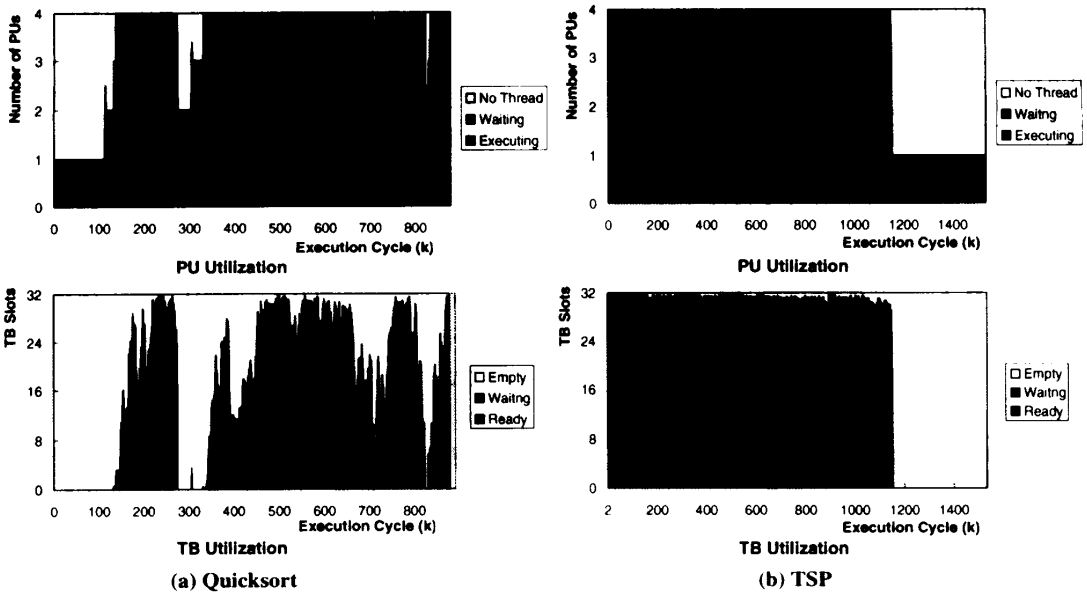

図9 クィックソートとTSPの実行プロファイル
Fig. 9 Execution profile of quicksort and TSP.

表3 メモリバス占有率の一覧
Table 3 Memory bus occupancy.

PU台数	Q.S.	FFT	TSP	N.-Q.	W.F.
1	8.5	21.7	15.8	9.8	8.1
2	16.4	43.0	29.0	20.5	18.1
4	28.3	76.4	48.8	39.0	35.8
6	33.8	93.2	61.5	57.1	52.4
8	39.9	95.1	68.3	73.3	68.2

性が低いことにあるが（データキャッシュヒット率は約65%程度），それに加えてナイーブなライトスループロトコルによりメモリバスのトラフィックが必要以上に増大していることがより直接的な要因となっていると思われる．

5.3 結果の分析

以上のような評価の結果，順序付きマルチスレッド実行モデルによるスレッドレベル並列処理の性能を低下させる要因として見い出されたのは（1）スレッドレベル並列性の欠如と（2）メモリネックの2つであった．これらの要因が存在しない場合は，本実行モデルによりきわめて実行効率の良いスレッドレベル並列処理を実現できることが分かった．特に過度のスレッドレベル並列性の動的な制限（N-クィーン），大量の同期動作の処理とそれにともなうスレッド切替え（ウェーブフロント）などが効率良く機能していることが確認された．特筆すべきことは，性能が飽和する場合でも4PU構成程度までは満足のいく性能向上率が得られている点と，8PU構成で最低でも4倍の性能向上が得られている点である．ナイーブなキャッシュモデル上での評価結果ということを考えあわせると，本実行モデルの効率の高さが十分に裏付けられたといえる．

4.1.2項で述べたように，本実行モデルにおけるスレッドスケジューリング手法にはいろいろなバリエーションが可能であり，その最適化によりさらに高い処理効率を実現することが可能になると考えられる．我々は，本稿で提案した手法のほかに，発行可能なTDのうちで最も仮想スレッド番号が小さいものや大きいものをつねに選択する，より簡単なスレッドスケジューリング手法の評価も別途行った．しかし，このような固定的な手法では再帰型のプログラムとループ型のプログラムの双方において高い効率を得ることはできなかった．これに対し，提案スケジュール手法はすべてのプログラムにおいてこれら固定的な手法で得られる最高の性能を上回る性能を示した．これは4.1.2項で述べたスレッドスケジューリングに関する予測の正当性を示唆していると思われる．

6. 議　　論

5章で明らかになった2つの性能律速要因は，順序付きマルチスレッド実行モデルの本質的な課題である．再度スーパースカラ実行モデルと対比すると，そこでの命令レベル並列性の限界とレジスタ多重アクセスのボトルネックという2つの律速要因が，そのままここでの2つの律速要因に対応すると見ることができる．すなわち，両モデルのように制御フローをベースとし

て実行時のオーバヘッドを少なくした並列処理を目指す限り，性能のスケーラビリティはある程度制約されざるをえない，ということができる．

このような観点から考えると，スレッドレベル並列性の利用のみではなく，命令レベル並列処理との融合的な処理形態が将来的には必須になると考えられる．これに関しては，すでにいくつかの研究[9]が報告されているが，これまでのところ，複雑なハードウェア機構を必要としアーキテクチャが重くなりがちだという欠点があり，より軽いハードウェアプリミティブを切り出すことが必要だと考えられる．また，メモリネックに関しては，よりスマートなキャッシュモデル/メモリモデルを用いることにより問題の軽減は可能であるが，より本質的にはデータの局所性をどのように管理するかという問題に帰着する．この点に関してはソフトウェア制御によるプリフェッチ技術などが有望であると考えられる．

スーパースカラ実行モデルにおいて単純な順序実行とより複雑な非順序実行が存在したように，順序付きマルチスレッド実行モデルにおいても，順序実行と非順序実行を考えることができる．この分類に従うと，1つのPU上での実行がつねに仮想制御フローに基づく逐次順序を忠実に遵守しているという点で，本稿で提案した実行モデルは順序実行にあたる．したがって，非順序実行を導入することにより本実行モデルの並列性抽出能力を増すことができると考えられる．このような非順序実行においては，レジスタリネーミング等のスーパースカラ技術から容易に類推されるように，駆動フレームを一時的にリネームする機構と一部の駆動フレームの生存情報を管理する機構が必要になる．

7. おわりに

本稿では，将来のマイクロプロセッサを想定したメモリ共有型の小規模並列計算機システムを対象とし，扱いやすくかつ効率の良いことを条件として，スレッドレベル並列処理を実現するための実行モデルおよびアーキテクチャを検討した．ここで「扱いやすい」とは，コードがスケーラブルである，コードがas isで実行可能である等の条件を指す．このような条件を満たすためには，同期ミス時のスレッド切替えが必要であるという点において，アーキテクチャサポートされたマルチスレッド実行モデルが必須となる．

一方従来のマルチスレッド実行モデルは，最大限の並列性を抽出する点に力点がおかれていたという点で，古典的なデータフローモデルと同じ立場に立脚していた．本稿で提案した順序付きマルチスレッド実行モデルは，上のような問題設定を踏まえ，スーパースカラ実行モデルと同じ立場，すなわち実行時のオーバヘッドを最小限に抑えた並列処理を目指す立場に立脚することを志したモデルである．本実行モデルによって，スレッドスケジューリングや駆動フレームの管理を簡単化することが可能となった．またシミュレーションによる評価結果においても，実際に本実行モデルが効率の良いスレッドレベル並列処理を実現することが確認された．

今後は，本実行モデルとそのアーキテクチャサポートのリファインを検討すると同時に，さらに大規模な評価を行う予定である．また，本稿では主としてアーキテクチャ側からのアプローチにより本実行モデルを提案したが，今後は並列プログラミングモデルからのアプローチを通して本実行モデルの検証を行うことが重要であると考えられる．このような逐次実行のセマンティクスを保ったスレッドレベル並列処理のプログラミングモデルについては未検討の課題が多く，興味深い研究領域を構成すると考えられる．

参考文献

1) Wall, D.W.: Limits of Instruction-Level Parallelism, *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.176–188 (1991).
2) Lam, M.S. and Wilson, R.P.: Limits of Control Flow on Parallelism, *Proc. 19th Annual International Symposium on Computer Architecture*, pp.46–57 (1992).
3) Sakai, S., Yamaguchi Y., Hiraki, K., Kodama, Y. and Yuba, T.: An Architecture of a Dataflow Single Chip Processor, *Proc. 16th Annual International Symposium on Computer Architecture*, pp.46–53 (1989).
4) Nikhil, R.S., Papadopoulos, G.M. and Arvind: *T: A Multithreaded Massively Parallel Architecture, *Proc. 19th Annual International Symposium on Computer Architecture*, pp.156-167 (1992).
5) Agarwal, A., Lim, B.-H., Kranz, D. and Kubiatowics, J.: April: A Processor Architecture for Multiprocessing, *Proc. 17th Annual International Symposium on Computer Architecture*, pp.104–114 (1990).
6) 井上俊明，本村真人，鳥居 淳，小長谷明彦：スレッドレベル並列処理アーキテクチャの検討，電子情報通信学会計算機アーキテクチャ研究会報告，No.107-11, pp.81–88 (1994).
7) Johnson, M.: *Superscalar Micoroprocessor*

Dessign, Chapter 6, pp.103–126, Prentice Hall (1990).
8) 宇佐見義之，加納義樹：巡回セールスマン問題の高速近似解法，*Computer Today*, No.63, pp.40–45 (1994).
9) Keckler, S.W. and Dally, W.J.: Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism, *Proc. 19th Annual International Symposium on Computer Architecture*, pp.202–213 (1992).

(平成7年8月31日受付)
(平成8年5月10日採録)

本村 真人

1962年生．1987年京都大学大学院理学研究科物理学第1専攻修士課程修了．同年NEC入社．1992年から93年にかけてMIT Laboratory for Computer Science客員研究員．NECマイクロエレクトロニクス研究所にて演算メモリ，マイクロプロセッサの研究に従事．1990年電子情報通信学会篠原学術奨励賞，1992年度IEEE Journal of Solid-State Circuits Best Paper Awordを受賞．電子情報通信学会，IEEE Computer Society各会員．

井上 俊明

1964年生．1989年電気通信大学大学院電気通信学研究科計算機科学専攻修士課程修了．同年NEC入社．現在マイクロエレクトロニクス研究所システムULSI研究部所属．LSIアーキテクチャの研究開発に従事．

鳥居 淳（正会員）

1967年生．1992年慶應義塾大学大学院理工学研究科修士課程修了．同年NEC入社．同社C&C研究所にて，マイクロプロセッサ，並列処理アーキテクチャ，並列処理プログラミングの研究に従事．

小長谷明彦（正会員）

1980年東京工業大学大学院修士課程情報科学専攻修了．同年NEC入社．1987年MIT Laboratory for Computer Science客員研究員．現NEC C&C研究所研究課長．博士（工学）．並列処理アーキテクチャ，並列分散OS，学習，遺伝的アルゴリズム，遺伝子情報処理に関心を持つ．人工知能学会，日本ソフトウェア科学会，電子情報通信学会，日本生物物理学会各会員．