■6 群（コンピュータ －基礎理論とハードウェア）－5 編（コンピュータアーキテクチャ（II）先進的）

2 章 スレッドレベル並列コンピュータ

（執筆者：佐藤寿倫）［2010 年 5 月 受領］

■概要■

現実的な命令レベル並列性の利用には限界が見えてきたこと，高いクロック速度は消費電力の問題を引き起こしたことなどを背景に，命令よりも粗い粒度の並列性を利用してコンピュータの性能向上を図るスレッドレベル並列コンピュータが注目されてきた．本章では，この背景から話を始め，いくつかのスレッドレベル並列コンピュータを紹介する．

【本章の構成】

スレッドレベルレベル並列コンピュータに主役が移った背景を理解するために，まず 2-1 節で命令レベル並列コンピュータの限界を解説する．スレッドレベル並列性を利用するマルチスレッドプロセッサとして，スループット指向コンピュータを 2-2 節で，投機を利用するレイテンシ指向並列コンピュータを 2-3 節で解説する．

 © 電子情報通信学会 *2010*

■6 群 － 5 編 － 2 章

2-1 命令レベル並列コンピュータの限界

（執筆者：小林良太郎）［2008 年 11 月 受領］

命令レベル並列コンピュータは，プログラムに内在する命令レベル並列（ILP）を抽出する．高い性能を得るためには，プログラム中に高い ILP が内在し，かつ，その ILP を抽出する能力（ILP 抽出能力）がコンピュータに備わっていなければならない．

そのため，近年の命令レベル並列コンピュータは，プログラムに内在する ILP の向上や，ILP 抽出能力の向上などを行ってきた．しかし現在のところ，両者はともに限界に達しており，これ以上の性能向上は困難になってきている．

本節では，まず，プログラムに内在する ILP の限界について述べる．次に，ILP 抽出能力の限界について述べる．これらを通じ，命令レベル並列コンピュータがいかにして限界に達したかを理解する．

2-1-1 プログラムに内在する ILP の限界

プログラムに内在する ILP は主に以下によって厳しく制限される．

・制御依存

・データ依存

・メモリアクセス

・メモリ曖昧性

以下，それぞれについて述べる．

(1)　制御依存

ある分岐命令に後続する命令は，当該分岐命令の結果が得られるまで実行できないとき，当該分岐命令に制御依存しているという．

制御依存は次の二つの原因により ILP を制限する．

第 1 に，分岐命令の処理を開始してからその実行結果が得られるまで後続命令を実行することができない．命令がパイプライン処理される場合，分岐命令をフェッチしてから実行するまでの間，後続命令の処理がストールする．これにより，ILP を引き出すことのできないサイクルが生まれる．

第 2 に，プログラムにおいて並列性を引き出すことのできる範囲が分岐命令間に制限される．マイクロプロセッサの最も一般的な応用である非数値計算プログラムにおいては，数命令に 1 回の頻度で分岐命令が現れるので，上記原因による ILP への制約は特に厳しい．

命令レベル並列コンピュータは，この制御依存を緩和するために，投機的実行を行っている．これは，分岐命令の実行結果が得られる前にその結果を予測し，分岐先の命令の処理を投機的に開始する技術である．投機的実行では，予測が成功した分岐に関する制御依存を取り除くことができる．分岐結果には偏りがあるため，それを利用することで，高い予測精度を得ることができる [6), 8)]．

分岐予測精度を上げれば，投機的実行の効果は更に大きくなるので，投機的実行を用いるプロセッサにおいて分岐予測精度の向上は非常に重要である．そのため，分岐予測に関する

研究が盛んに行われてきたが，予測精度の向上はほぼ飽和しており，制御依存の緩和もほぼ限界に達したといえる．

(2) データ依存

ある命令の生成したデータを後続命令が使用するとき，両者はデータ依存関係にある．真のデータ依存とも呼ばれる．データ依存関係にある命令は並列に実行できないため，ILP を制限する要因となる．しかし，それだけではデータ依存がどの程度並列性に影響を与えるか直感的に理解することはできない．

そこで，命令をノードとし，データ依存関係にある命令を，データの生成元から使用先へと向かう辺で結んだデータフローグラフ（DFG）を考える．なお，簡単化のため，ここでは命令の実行サイクル数は常に同一とし，データ依存以外に制約は存在しないと仮定する．

DFG において，有向辺で結ばれたパス上にある命令は，直接的あるいは間接的にデータ依存関係にあるため，逐次的に実行していく必要がある．したがって，DFG のパスにおいて，最も長いパス（クリティカルパス）がプログラムの実行時間，ひいては，ILP を決定するということになる．非数値計算プログラムなどにおいては，クリティカルパスが長くなり，ILP を厳しく制限する傾向がある．

データ依存を緩和する技術の一つとして，値予測 [1), 7), 10)] がある．これは，命令の実行結果が得られる前にその結果を予測するという技術である．値予測によって，命令の実行結果が予測できれば，その値を使用する命令の処理を投機的に開始することができる．予測が成功した場合，命令間のデータ依存を解消することができる．しかし，分岐予測に比べ，値予測による ILP 向上効果は低い．その原因として以下の二つをあげることができる．

第 1 に，値予測の精度は低い．プログラムに依存するが，値予測は分岐予測よりもかなり低い精度しか達成できない．これは，値予測では，予測対象が多数存在し，かつ，過去に生成した値の履歴と，これから生成する値との相関が低いからである．

第 2 に，データ依存の解消は必ずしも ILP の向上に貢献しない．先ほど示したように，プログラムの ILP は，DFG におけるクリティカルパスによって決まるため，クリティカルパス上にない命令のデータ依存が解消されても ILP は向上しない．また，データ依存の解消によって，クリティカルパスが短くなるにつれて，クリティカルパス上にない命令の数が増えていくため，ILP の向上はますます難しくなっていく．

値予測の効果を改善するには，クリティカルパス上の命令の値予測精度を上げる必要があるが，値予測精度はほぼ飽和しており，データ依存の緩和もほぼ限界に達したといえる．

(3) メモリアクセス

先ほど示した DFG では，簡単化のため，命令の実行サイクル数は常に同一としていた．しかし，実行サイクル数は，命令の種類などによって決まる．

通常，ロード／ストア命令の実行サイクル数が最も大きい．この原因は，メモリアクセスにある．ハイエンドのマイクロプロセッサでは，メモリアクセスが数百サイクルに達する場合があり，算術演算に要するサイクル数をはるかに上回る．また，ロード／ストア命令は基本的な命令であり，メモリアクセスは頻繁に行われる．これらより，DFG におけるクリティカルパスは極端に長くなり，ILP が非常に厳しく制限される．

メモリアクセスを高速化する効果的な技術としてキャッシュがある．キャッシュは，メモリよりも小容量で高速なバッファである．一部のメモリデータしか保持できないが，ロード／ストア命令のアクセスするデータがキャッシュ上に存在（ヒット）すれば，高速にデータを得ることができる．データアクセスには局所性があるため，それをうまく利用することで，高いヒット率を得ることができる．

キャッシュヒット率を上げれば，キャッシュの効果は更に大きくなるため，キャッシュの構成や更新に関する研究が盛んに行われてきたが，分岐予測や値予測と同様に，キャッシュヒット率の向上もほぼ飽和している．

キャッシュのほかに，メモリアクセスを高速化する重要な技術として，プリフェッチ [3), 4)] がある．プリフェッチは，ロード／ストア命令がアクセスする可能性が高いと考えられるデータを予めフェッチしておく技術である．メモリアクセスのパターンによっては，非常に効果的に動作するが，その改善もほぼ限界に達している．

このようにメモリアクセスの高速化が限界に達する一方で，メモリとプロセッサの速度差は広がり続けており，今後，メモリアクセスの影響は，より大きくなっていく．

(4) メモリ曖昧性

メモリ曖昧性とは，アクセスするメモリアドレスを計算するまで，ロード／ストア命令間のデータ依存関係がわからないことをいう．より正確には，あるロード命令のメモリアドレスが計算され，かつ，それに先行する全ストア命令のメモリアドレスが計算されたときに，当該ロード命令とその先行ストア命令のデータ依存関係が判明することを示す．

メモリ曖昧性により，ロード命令は，自身のメモリアドレスの計算が終了していても，先行する全ストア命令のメモリアドレスが計算されるまで，メモリアクセスを行うことができない．これにより，ILP が厳しく制限される．

これまでにメモリ曖昧性を緩和するための技術がいくつも提案されている．代表的な技術の一つとして，メモリ依存予測 [2), 5)] がある．これは，メモリアドレスを計算する前に，ロード命令とストア命令のデータ依存関係を予測し，依存がないと予測したロード命令については，投機的にメモリアクセスを行う技術である．メモリ依存予測の精度は非常に高く，メモリ曖昧性が ILP に与える影響をほぼ取り除くことができる．

2-1-2 ILP 抽出能力の限界

命令レベル並列コンピュータは，ILP 抽出能力を向上させるため，発行幅の拡大を行ってきた．一方，発行幅の拡大は，コンピュータの各構成要素（結果バス，レジスタファイル，ROB など）の規模拡大をまねく．これにより，コンピュータ内の配線長やゲート数が増加してしまうが，微細化によってゲート遅延と配線遅延の減少が実現できていたため，結果として，コンピュータの動作周波数は向上し続けてきた．

しかし近年では，微細化に対し，ゲート遅延の減少は従来と同様であるが，配線遅延の減少は緩やかである．コンピュータの構成要素には，結果バスのように，遅延時間が長く，かつ，長い配線をもつものがいくつか存在する．これらは，配線遅延が支配的であるため，微細化の恩恵を受けることができず，コンピュータの動作周波数に悪影響を及ぼす [9)]．通常，発行幅が拡大するにつれ，プロセッサの規模は急速に拡大していくため，配線遅延が動作周

波数に及ぼす悪影響も急速に増大していく．その結果，ILP 抽出能力は限界に達する．

■参考文献

1) T. F. Chen and J. L. Baer, "Effective Hardware-based Data Prefetching for High Performance Processors," IEEE Transactions on Computers, vol.44, pp.609-623, 1995.
2) G. Z. Chrysos and J. S. Emer, "Memory Dependence Prediction using Store Sets," In Proc. 25th Int. Symp. on Computer Architecture, pp.142-153, 1998.
3) D. Joseph and D. Grunwald, "Prefetching using Markov Predictors," In Proc. 24th Int. Symp. on Computer Architecture, pp.252-263, 1997.
4) N. P. Jouppi, "Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers," In Proc. 17th Int. Symp. on Computer Architecture, pp.364-373, 1990.
5) R. E. Kessler, E. J. McLellan, and D. A. Webb, "The Alpha 21264 Microprocessor Architecture," In Proc. Int. Conf. on Computer Design, pp.90-95, 1998.
6) J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," IEEE Computer, vol.17, no.1, pp.6-22, 1984.
7) M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," In Proc. Seventh Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp.138-147, 1996.
8) S. McFarling, "Combining Branch Predictors," WRL Technical Note, TN-36, Digital Equipment Corporation, 1993.
9) S. Palacharla, N. P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," In Proc. 24th Int. Symp. on Computer Architecture, pp.206-218, 1997.
10) Y. Sazeides and J. E. Smith, "The Predictability of Data Values," In Proc. 30th Int. Symp. on Microarchitecture, pp.248-258, 1997.

■6 群 － 5 編 － 2 章

2-2 スループット指向コンピュータ

（執筆者：中條拓伯）［2008 年 11 月 受領］

前節で述べたように，命令レベル並列性の抽出には限界があり，さらなる性能向上を目指すためには，処理単位，すなわち計算粒度の大きなものを単位として，それらを並列に処理する方向が有効であると考えられている．

複数の命令をひとまとめとして扱う処理の単位としてスレッド（Thread）に着目し，そのスレッド単位で明示的（Explicit）に並列処理を行うか，暗黙的（Implicit）に行うかによって大きく二つに分類し，前者をスレッドレベル並列処理，そして後者をスループット指向並列処理と定義する．

本節ではスループット指向並列コンピュータ [1)]について述べ，スレッドレベル並列処理については，次節で詳しく述べてある．

2-2-1 複数スレッド処理とスループット指向コンピュータ

複数のスレッドを，一つのコンピュータシステム上で同時に実行することができれば，単位時間で処理できる演算量，すなわちスループットを向上させることが期待できる．

複数スレッドを同時に実行する処理形態を広い意味でマルチスレッドプロセッサと呼ぶ．マルチスレッドプロセッサは，主としてレジスタと PC からなるスレッドのコンテキストを複数保持し，そのコンテキストを自分で切り替える．スレッドを切り替えるタイミングによって，更に粗粒度マルチスレッドと細粒度マルチスレッドに分類され，前者はキャッシュミスや TLB ミスが生じたときなど，長時間プロセッサをストールさせるような場合を契機とするといった，切り替え単位が比較的大きなものを示し，後者は数命令，もしくはサイクルごとにスイッチするといった短時間でスレッドを切り替えるものである．

また単一チップ上においてスレッド処理を行うには，複数のコアを有するマルチコアプロセッサ上においてそれぞれスレッド処理を行う場合と，単一コアの中で複数のスレッドを同時実行する場合がある．後者の場合，スレッドは演算器といった計算リソースを常に使用しているわけではなく，例えば整数演算を行っているときには，浮動小数点演算ユニットはフリーな状態にあり，そのときは他のスレッドが使うことができ，計算リソースを有効活用できることとなる．この両者を含めてシングルチップ上で行うことから，チップマルチスレッドと呼ぶこともある．

このように，単一時間内の演算処理量を向上させるものをスループット指向コンピュータと定義する．

各マルチスレッドプロセッサの概念を**図 2･1** に示す．縦軸はクロックサイクルを，各ブロックは計算リソースとしての演算器を表す．図 2･1 において，プロセッサは四つの演算器をもつと仮定する．

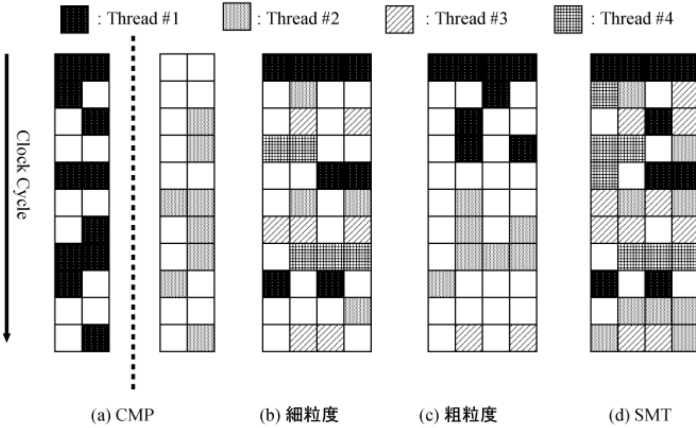

図 2・1　各マルチスレッドプロセッサの概念図

(1)　チップマルチプロセッサ（CMP）によるマルチスレッド処理

チップマルチプロセッサ（CMP）は，マルチコアプロセッサとも呼ばれ，マルチスレッドに対応するため，1 チップ内に複数のプロセッサコアを実装している．

演算器やレジスタなどの計算リソースは各コアで独立して保持する．2 コアの CMP によるスレッド処理の概念を図 2･1(a) に示す．演算器は各コアにそれぞれ専用に割り当てられており，一つのスレッドがコアをまたがって実行することはできない．実行するスレッド数を増加させて性能向上を図るには，CMP ではコアを新たに追加しなければならない．

(2)　細粒度マルチスレッドと粗粒度マルチスレッド

細粒度マルチスレッドはサイクルごとに実行スレッドの切り替えを行い，マルチスレッドに対応している．図 2･1(b) にその概念を示す．近年の商用マシンでは，コア内で 4 スレッド同時実行を可能とし，8 コア搭載により，最大 32 スレッドまで同時実行可能なものもある．

粗粒度マルチスレッドは，時分割でスレッド切り替えを行うが，細粒度と異なり，何らかの契機により実行スレッドを切り替える．図 2･1(c) にその概念を示す．スレッド切り替えの契機として，キャッシュミスなど長時間プロセッサをストールしなければならない要因を採用する場合が多い．

細粒度方式，粗粒度方式はシングルスレッドプロセッサのコンテキスト切り替えとは異なり，いくつかのスレッドコンテキストをプロセッサ内部に保持して，スループットを向上させている．

細粒度方式の利点として，メインとなるスレッドの性能を低下させないことがあげられる．粗粒度方式では，強制的に数サイクルおきにスレッドを切り替えるため，プロセッサ全体のスループット性能は向上するが，スレッド当たりの処理性能は低下する場合があり，特定のスレッドの処理を高速化することができない．しかしながら，スレッドに優先順位をもたせ，実時間処理に対応させるといった研究事例もある．

一方，粗粒度方式はメインとなるスレッドがストールした場合のみ，他のスレッドを実行

すれば，メインスレッドの処理性能は低下しない．そのため，ある特定のスレッドを高速化したい場合に使用される．

ここでは，スループット指向コンピュータの一つとして，同時マルチスレッド（SMT：Simultaneous Multi-Threading）プロセッサ [2] について詳しく述べる．

2-2-2 同時マルチスレッドコンピュータ

同時マルチスレッド（SMT）コンピュータは，何らかの契機でスレッドを切り替えるのではなく，複数のスレッドを物理的に同時に実行するものである．SMT は，命令を単位としてスレッドを切り替えていると考えることもでき，細粒度マルチスレッドより更に細粒度であるといえる．

SMT は，既存のアウトオブオーダ・スーパースカラプロセッサをベースに容易にマルチスレッド化が可能である．すなわち，スレッドコンテキストを複数保持できるようにするほかは，ほとんど単に複数スレッドからフェッチした命令を命令ウィンドウ中に混在させるだけでよい．したがって，制御ロジックを若干付加するだけで，複数スレッドから実行可能な命令を選択し，演算器に発行することができる．このように SMT は，従来のマルチスレッドよりスレッドの粒度が小さいため，計算リソースの利用率を高め，スループットを向上しやすくする．

図 2･1(d) に SMT プロセッサの概念を示す．SMT は CMP とは異なり，各種演算器やレジスタファイル，キャッシュメモリなど多くの計算リソースをスレッド間で共有し，リソースのアイドル状態をできる限り回避してスループットの向上を図るのである．そのため，CMP と比べると少ないハードウェアリソースで実行スレッド数を増やすことができる．

また，細粒度，粗粒度方式と比較して，SMT プロセッサは時分割でスレッドを実行するのではなく，同タイミングにおいても空きリソースが存在すれば，複数のスレッドを実行できる．この特徴により，同じスレッド数における実行性能を比較した場合，SMT プロセッサは細粒度，粗粒度に比べ，高いスループット性能を発揮できる可能性がある．

2-2-3 スループット指向コンピュータの限界

スループット指向コンピュータには，以下のような問題点がある．

- **レジスタファイル容量の増大**

  スループット指向コンピュータでは，スレッド数に応じてレジスタ数を増加させる必要があるが，レジスタ数を増やすことは，そこにホットスポットを生成することとなり，スループットの低下を招くこととなる．このため，スループット指向コンピュータでは，サポートするスレッド数とシングルスレッドの性能／効率との間に，トレードオフが発生することになる．

- **乏しいスケーラビリティ**

  スループット指向コンピュータのポイントは計算リソースのアイドル状態を回避することにあるため，実装しているリソースを超えるスレッド数を増やしても性能は頭打ちとなる．実行するスレッド数によっては，共有しているリソースの競合や枯渇が発生し，性能低下を引き起こしてしまう事態も考えられ，リソースの数を増やすことで解決できるものではない．現状では，スレッド数は通常 2，多くても 4 程度が適切な数であ

ると考えられている．

- **キャッシュヒット率の低下**

複数のスレッドを平行して実行するため，ワーキングセットサイズが増大し，結果として，キャッシュヒット率が低下する傾向にある．マルチスレッド環境でキャッシュヒット率の維持・向上を図るには，より大容量なキャッシュメモリが必要となる．更に，スレッド間でキャッシュメモリを共有する場合は，スレッド間のキャッシュライン競合によるミスが多発し，性能が低下する．

マルチスレッド環境において，同時に実行するスレッドのメモリアクセスの類似性に着目してスケジューリングに工夫を施したり[3)]，アクセスの傾向からキャッシュの構成を切り替えるといった提案[4)] もある．

以上から，スループット指向コンピュータは高い性能を示す可能性を秘めながら，必ずしも理想的な性能を示すものではない．

2-2-4 スループット指向コンピュータの実用化

スループット指向コンピュータでは，スレッド数とシングルスレッドの性能／効率のトレードオフのため，実用化には利用形態について慎重になる必要がある．同時に実行すべきプログラムが多くない場合には，スループット指向コンピュータは有効ではなく，その場合は，次節のスレッドレベル並列処理に託すこととなる．

サーバなど，定常的に複数のプログラムを実行する環境おいて，スループット指向コンピュータが威力を発揮する．キャッシュミスのレイテンシをマルチスレッドによって隠ぺいする商用サーバもあり，プリフェッチが有効でないプログラムに対して極めて有効な手段となる．

数値演算やメディア処理など，データ並列性をもつプログラムでは比較的容易にマルチスレッド化が可能であり，スループット指向コンピュータで実行する方向性もある．しかしこれらのプログラムでは，もともとリソースの利用率が高く，また処理の類似性からリソースの競合が発生する可能性は高い．しかしながら，電力効率が最重要視されるサーバでは，このような方向性が有効である可能性がある．

■参考文献

1) 五島正裕, “スーパスカラ/VLIW プロセッサとスループット指向 MT プロセッサ,” 情報処理学会誌, vol.46, no.10, pp.1104-1110, 2005.
2) D. Tullsen, S. Eggers, and H. Levy, “Simultaneous multithreading: Maximizing on-chip parallelism,” In Proc. of the 22th International Symposium on Computer Architecture (ISCA’1995), pp.392-403, 1995.
3) 内倉　要, 笹田耕一, 佐藤未来子, 加藤義人, 大和仁典, 中條拓伯, 並木美太郎, “SMT プロセッサにおけるスレッドスケジューラの開発,” 情報処理学会論文誌, vol.46, no.SIG12 (ACS11), pp.150-160, 2005.
4) 小笠原嘉泰, 佐藤未来子, 並木美太郎, 中條拓伯, “SMT プロセッサにおけるキャッシュリプレース動的切替え方式,” 情報処理学会論文誌, vol.48, no.SIG13 (ACS19), pp.70-83, 2007.

■6 群 － 5 編 － 2 章

2-3 レイテンシ指向並列コンピュータ

（執筆者：大津金光）［2008 年 10 月 受領］

スレッドレベル並列性を活用して高性能化を達成するコンピュータとしてはマルチスレッドプロセッサとマルチコアプロセッサ（あるいはチップマルチプロセッサとも呼ばれる）があげられる．両者は非常に似た概念であるが，厳密には区別される．マルチコアプロセッサは独立性の高い複数個のプロセッサコアがオンチップあるいはオンダイに集積されているプロセッサ形態を指すのに対し，マルチスレッドプロセッサはハードウェアにより複数のスレッドを制御・管理できるプロセッサ形態を指す．両者は互いに直交する概念であるため，マルチコアプロセッサでありマルチスレッドプロセッサであるというものも存在する．

マルチスレッドプロセッサはスレッドレベル並列性の利用の仕方によって明示的マルチスレッディング（Explicit MultiThreading）と暗黙的マルチスレッディング（Implicit MultiThreading）とに大別できる [1]．前者はプログラム中で明示的に（Explicit に）現われるスレッドレベル並列性を活用するもので，マルチプログラム環境や（OS 的定義による）マルチスレッドプログラムの性能を高めるものである．後者は逐次プログラム中に潜むスレッドレベル並列性を活用して，シングルプログラムの性能を高めるものである．

更に，明示的マルチスレッディングはスレッド切り替えの条件によって，インターリーブ型マルチスレッディング（IMT : Interleaved MultiThreading），ブロック型マルチスレッディング（BMT : Blocked MultiThreading），同時マルチスレッディング（SMT : Simultaneous MultiThreading）の三つの形態に大別される．IMT は細粒度マルチスレッディング（FGMT : Fine-Grained MultiThreading）とも呼ばれ，サイクルごとに実行されるスレッドが切り替わるという実行方式である．BMT は粗粒度マルチスレッディング（CGMT : Coarse-Grained MultiThreading）とも呼ばれ，キャッシュミスなどのレイテンシの大きいイベントが発生した際にスレッドが切り替わるという実行方式である．SMT はスーパースカラプロセッサを前提として，複数のスレッドの中から実行可能な命令を選択して同時に実行するという実行方式である．

明示的マルチスレッディングは逐次シングルプログラムの高速化（すなわち，実行時間の短縮）というよりは，むしろ，単位時間にできるだけ多くの処理を行うスループット指向の実行方式であると考えることができる．これに基づいたコンピュータについては前節において解説済みであるので，本節では，暗黙的マルチスレッディングに分類されるスレッドレベル並列コンピュータについて述べる．

なお，暗黙的マルチスレッディングに基づいたスレッドレベル並列コンピュータは，執筆時点で一部商用化されているもの [2] があるが，ほとんどは研究レベルのものである．

2-3-1 投機的マルチスレッディング

逐次のシングルプログラムの実行を高速化する手段として，従来よりプログラム中の命令レベル並列性が活用されてきた．現在のマイクロプロセッサにおいては，スーパースカラや VLIW 方式のものがごく一般的なものとなっている．しかしながら，現実的には命令レベルの並列度は高くないため，これまで以上に劇的な高速化は望めない状況にある．暗黙的マル

チスレッディングは，逐次のシングルプログラムの中からスレッドレベルの並列性を活用して高速化を図るものである．

逐次のプログラムからスレッドレベル並列性を抽出する場合，プログラムに内在する制御依存やデータ依存によって，スレッド間での並列実行が阻害される場合が多い．スレッド間で保守的な同期・通信を行っていたのでは性能が上がらないため，スレッドレベルでの投機実行，すなわち投機的マルチスレッディングが導入される場合が多い．

投機的マルチスレッディングは並列実行の一形態で，逐次のシングルプログラムを複数の小さいスレッドに分割し，将来実行される可能性が高い場合や，スレッド間での依存データの受信待ちのスレッドにおいて将来受信されるデータ値が予測できるような場合に，前倒しで並列にスレッドの実行を行うものである．もし前倒しの実行が成功すれば，前倒した分だけ全体の処理を早期に完了することになる．しかし，前倒しの実行が失敗した場合，すなわち（投機）スレッドを実行してはいけなかった場合には，そのスレッドによって変更を受けたレジスタやメモリの状態をスレッドの実行開始時点の状態まで一旦回復した後，本来実行すべきであったコードを実行し直すことで，元の逐次のシングルプログラムの実行結果と同じになるように補償する必要がある．そのため，全体の処理の完了が前倒ししない場合よりも遅れてしまう危険性もある．ただし，実際のプログラムの実行には制御やデータについて「偏り」や「局所性」が存在するため，それを活用することで投機的なスレッド実行が成功する場面も多い．そのため，投機的マルチスレッディングはプログラムの実行における「偏り」や「局所性」を積極的に活用してスレッドレベル並列実行に転化し，高速化を図る実行方式と解釈することができる．

2-3-2 分類と特徴

投機的マルチスレッディング方式として様々な方式が考えられているが，以下の点に注目することで分類することができる．

1. スレッド分割の手段
2. スレッド処理ユニットの共有形態
3. スレッドの割り当て方法
4. レジスタデータの扱い
5. メモリデータの扱い

1. について，スレッドの分割をコンパイラなどのソフトウェアによって（もしくは人手によって）行うもの [2)~6), 11)] と，プログラムを実行しながらハードウェアで自動的に分割していくもの [7)~10)] とがある．

前者は，コンパイラ（あるいは人手）によって元のプログラムをマルチスレッド化したプログラムに作り変えることが必要である．ソフトウェアにより広範なコード解析を行うことができるため，性能の高いマルチスレッド化プログラムにできる可能性が高い．しかし，この方式の場合，（一般のコンパイラの最適化と同様に）実際のプログラムの挙動が，スレッド分割の際に想定したシナリオと異なっている場合には，性能が高くならない（あるいは低下する）場合もあり得る．

後者は，ハードウェアがプログラムの実行を進めながら実行の様子を監視し，マルチスレッド化の条件がそろったところで，スレッド分割を行う．スレッド分割にはスレッド実行

の開始が伴うことが多い．マルチスレッド化の条件としては，ループの先頭の検出や，手続き呼び出しの検出などがある．ループの先頭を検出した場合は，ループの各イテレーションをスレッドとして並列実行を行うようにスレッド分割を行い，マルチスレッド実行を行う．手続き呼び出しを検出した場合には，呼び出し元の処理と呼び出し先の処理を別スレッドとして分割し，マルチスレッド実行を行う．ハードウェアによるコード解析能力はソフトウェアでの解析に比べると低いため，コードの性能自体は劣る可能性があるが，実行中の挙動を素早くマルチスレッド実行に反映することができるため, 全体の性能は高くなる場合もある．

2. について，複数の（投機的）スレッドが実行処理される際に使用される計算資源に関して，スレッドを実行する処理ユニット（以降，スレッド処理ユニット）がスレッドごとに独立しているもの [3), 4), 6)~8), 10)] と，スレッド間で共有されるもの [9), 11)] とがある．スレッドごとに独立している場合は，ハードウェア的にはマルチコアプロセッサと同様のものとなり，各スレッドはプロセッサコア 1 個分の計算資源をすべて利用することができるメリットがある．その一方で，スレッド実行中に「待ち」が発生した場合には，その間の計算資源が利用されないままとなり，利用効率が低下するデメリットも併せもつ．スレッド間で共有している場合は，一つのスレッドが休止していても他のスレッドの実行に計算資源が利用されるため，利用効率は低下しない．しかし，資源を共有していることにより，各スレッドの実行に要する時間は長くなる可能性がある．

3. について，前項の 2. と大きく関係するが，スレッド処理ユニットがスレッドごとに独立している場合，スレッドをどの処理ユニットに割り当てるかが問題となる．投機的マルチスレッディングでは，各スレッドはプログラム順に実行が開始されるため，これをスレッド処理ユニット間の接続形態に反映することで,（プログラム順に実行される）スレッドの割り当て方法とハードウェア的な接続形態が一致することになり，スレッドの割り当て処理を簡潔にできる．例えば，プログラム中の制御フロー上の 1 本のパスに絞って，スレッドを並列実行していく場合，スレッドは 1 次元上に整列するので，スレッド処理ユニット間を単方向のリング接続すれば，自然にスレッドの割り当て方が決まる．

このように，スレッド間の先行関係とスレッド処理ユニット間の接続関係が対応しているもの [3), 4), 8)] と，対応していないもの [7)] とがある．前述のとおり，前者はスレッド割り当てはスレッド処理ユニット間の接続形態によって自然に実現されているために，割り当てのためのロジックを要しない．しかしその反面，スレッドの割り当て方が一意に決まってしまうため，スレッド間での負荷が均等でない場合に，スレッドを実行する処理ユニットに無駄な空きが生じて計算資源の利用効率が低くなる場合がある．後者はプログラム順とは無関係に空いているスレッド処理ユニットにスレッドを割り当てることができるので，計算資源の利用効率が低くなりにくいが，スレッド割り当てのためのハードウェアが必要になる．

4. については，まず，スレッド間でレジスタのイメージを共有するかしないかで分類される．投機的マルチスレッディングでは，逐次のシングルプログラムがベースとして存在するため，その命令コードをそのまま利用する形でスレッドコードを生成した方が都合がよい場合も多い. この場合, レジスタのイメージを共有していた方がスレッドコードを作りやすい. また，高速なスレッド間通信をレジスタを介して行う方が性能上有利となる．

各スレッドでレジスタのイメージを共有する場合は，プログラム順にレジスタのデータ依存関係が発生することになるが，この依存にどう対応するかで選択肢がある．まず，最も保

守的な選択として同期通信によって依存するデータを転送する方法がある．スレッド実行におけるレジスタごとの最終値を（プログラム順のうえでの）後続のスレッドに転送されなければならないが，このとき後続のスレッドは先行するスレッドからのデータが到着するまで実行を中断し，データの到着後に実行を再開することで，正しい実行を保証する．

別の選択肢として，後続のスレッドが先行スレッドから送られてくるデータの到着を待たないという方法がある．これは後続のスレッドが先行スレッドからのデータを待つのではなく，到着するであろうデータ値を予測して，処理を前倒しで進めてしまうというものである．後で先行スレッドからデータが到着した際に，予測した値と比較を行い，もし両者が一致すれば，前倒しで実行した結果をそのまま採用することで，実行時間が短縮される．もし一致しなかった場合は，実行結果は正しくないため棄却され，実行をやり直すことになる．このような実行方法を「値予測」に基づいた「データ投機実行」と呼ぶ．

更に，ハードウェアでスレッド分割を行う実行方式の場合には，ハードウェアのコード解析能力の制約により，前もってスレッド間でのレジスタの依存関係があるかどうかがわからない状況が存在するため，取り敢えず依存がないものとして実行を進めておき，後で依存があることが判明した時点で実行をやり直すという方法をとることがある．これも「データ投機実行」の一つである．これの発展形として，スレッド間で依存があるかどうかを予測して，依存があると予測した場合のみ同期通信を行う方法もある．これは「依存予測」に基づいた「データ投機実行」である．

5. については，レジスタデータの扱いと共通する部分も多いが，全く同じではない．メモリデータとレジスタデータの扱いにおいて決定的に異なるのは，間接アクセスがあるかどうかである．スレッド間でのレジスタの依存関係はレジスタの番号の比較によって把握することができる．しかしながら，メモリデータはポインタを介した間接アクセスが存在するため，静的なコード解析だけでは依存関係があるかどうかを完全に判定することはできない．そのため，メモリデータの依存関係をきちんと守ろうとすると，どうしても保守的な同期通信処理を行う必要があるため，スレッド間での並列実行が阻害される結果になる．そこで，メモリデータに関して保守的に同期通信を行うのではなく，依存予測あるいは値予測を行ってデータ投機実行を行うものも提案されている．

以上，投機的マルチスレッディング方式がいくつかの視点により分類できることを示してきた．次に，投機的マルチスレッディングの具体的な事例について主なものをいくつか紹介する．

2-3-3 Multiscalar Processor

Multiscalar Processor アーキテクチャ [3] は投機的マルチスレッディングの草分け的存在である．Multiscalar では，プログラムの制御フローグラフを静的に分割してできるスレッドコード「タスク」をスレッド処理ユニット上で並列に実行する．この際，各タスクはプログラム順に実行される．最も先行する（したがって，最も古く実行を開始した）スレッドは head thread と呼ばれ，実行は非投機状態（すなわち，確定状態）である．head thread 以降のスレッドはすべて投機状態であり，スレッド間での依存違反発生により実行がキャンセルされる可能性があるが，head thread は実行がキャンセルされることはない．スレッド間のデータ通信はプログラム順と同一方向への通信のみであり，後続の（新しい）スレッドから先行の（古い）スレッドの方向へのデータ通信は発生しない．

図 2･2 に Multiscalar Processor のアーキテクチャを示す．スレッドを実行する Processing Unit が単方向のリングネットワーク上に複数個接続されている構成をとる．Processing Unit は L1 キャッシュ，プロセッサコアから構成され，スレッド間での単方向通信が行えるレジスタファイルを備えている．Processing Unit は Sequencer と呼ばれる全体のスレッド制御・管理を行うコンポーネントと接続されており，各スレッドは Sequencer からの指示によって Processing Unit 上で実行が開始される．スレッドはリングネットワークの順方向に一つずつ生成され，これによって投機的マルチスレッディングが実現される．Multiscalar における（マルチスレッド化された）プログラムでは，機械命令コードとともに，task descriptor と呼ばれるタスク間の遷移先情報とタスクが値を変更する可能性のあるレジスタ番号の集合を格納したデータ構造を用意する．Sequencer は task decsriptor 内に記述されているタスク間の遷移先情報に基づいて，次に実行される可能性の高い遷移先のタスクを空いている Processing Unit を割り当てスレッドの実行を開始させる．

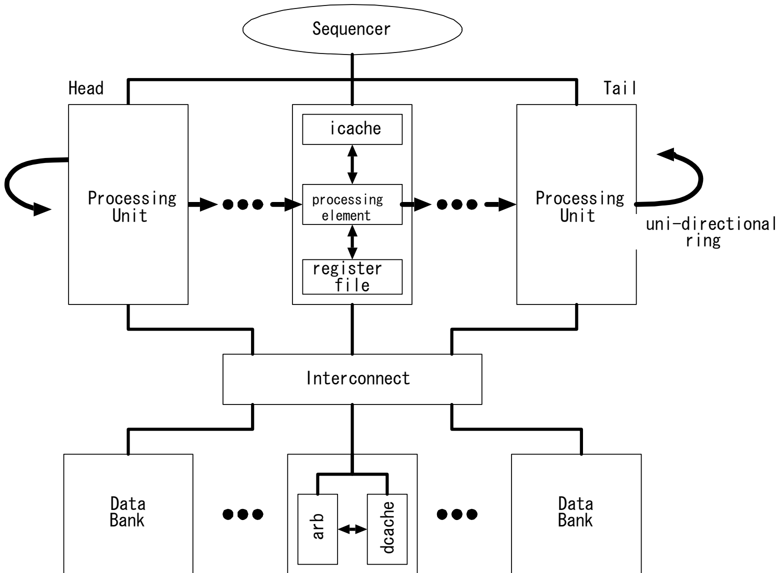

図 2・2　Multiscalar processor [3)]

各タスクの実行開始は Sequencer によって制御されるが，実行の終了については命令コードによって行う．Multiscalar では一般的なプロセッサの機械命令セットをベースに拡張を施し，各命令ごとに制御用のタグビットを付加できるようになっている．タスクの終了を指定するためのタグビットとして，stop bit が用意されており，タスクの最後の命令にこのビットを付加することで，タスクの終了を指定する．

各 Processing Unit 内のレジスタファイルは後続のスレッドに対してデータを送信する機能をもち，送信を行うタイミングの制御は命令コードによって行う．前述の制御用のタグビットの一つとして，レジスタの転送のタイミングを指定するための forward bit が用意されており，これが付加された命令が実行された際にスレッド間でレジスタデータの転送が開始され，

目的レジスタの値を後続スレッドの同一番号のレジスタに転送する．これによって，各スレッド間でプログラム順に同じレジスタのイメージが参照されるようになる．

Processing Unit はメモリアクセス用のネットワークを経由して，Data Bank につながっている．Data Bank はデータキャッシュと ARB（Address Resolution Buffer）から構成される．投機状態にあるスレッドはその実行がキャンセルされる可能性があるため，メモリにストアしたデータを実行が確定状態になるまで実際にメモリに反映するわけにはいかない．そのため，Multiscalar では ARB と呼ばれる投機状態のメモリストアデータを格納するバッファを用意している．ARB は投機状態のスレッドから発行されたストアアクセスを格納し，ストアしたスレッドの後続のスレッドからロード要求があった場合に，対応する値を返す役割を果たす．また，メモリデータにおける依存違反のチェックも行っており，後続のスレッドが真に依存関係にあるストアが実行される前に（投機的に）ロードを行った場合に，依存違反を検出し，誤ったロードを行ったスレッドの実行のやり直しを要求する．

Multiscalar について特徴をまとめる．まず，スレッドの分割についてはコンパイラによって静的に行われる．コンパイラによって静的に生成された task descriptor に記述した内容により，Sequencer がスレッドを生成する．各スレッドは，リング接続された Processing Unit 上で独立して実行される．また，リング接続という形でスレッド（タスク）の割り当て方は一意に決まっている．スレッド間でレジスタのイメージを共有する方式をとっており，レジスタのデータ依存関係については，task descriptor 内の情報と forward bit 情報によって同期通信を行う．最後に，メモリデータについては，後続のスレッドは先行スレッドから送られてくるデータの到着を待たずにデータ投機実行を行う方式を採用している．

2-3-4 Superthreaded Architecture

Superthreaded Architecture[4] はコンパイラの並列化能力を最大限に活用することを前提として，ハードウェア構成の簡素化を図ったものである．前述の Multiscalar と同様に，プログラムの制御フローグラフを静的に分割してタスクを形成し，それらをスレッド処理ユニット上で並列実行する．Multiscalar との違いは，スレッドの実行開始の制御方法と，メモリデータの扱いである．Superthreaded Architecture では，スレッドの実行を集中的な Sequencer によって起動するのではなく，各スレッドコード内に埋め込まれたスレッド生成命令によって制御する．そのため，各スレッドは自分の次に実行されるスレッドの開始位置を指定することができる．Multiscalar と同様に，スレッド間のデータ通信はプログラム順と同一方向への通信のみであり，後続の（新しい）スレッドから先行の（古い）スレッドの方向へのデータ通信は発生しない．Multiscalar と異なり，本アーキテクチャではスレッド間通信はメモリを介したものだけが可能であり，レジスタデータの直接通信はできない．また，メモリデータについては，Multiscalar と違い，データ投機を行わない．

図 2･3 に Superthreaded Architecture を示す．Multiscalar と同様に，スレッドを実行する Thread Processing Unit が単方向のリングネットワークでつながっている．Thread Processing Unit は命令キャッシュとデータキャッシュを共有している．Thread Processing Unit は汎用プロセッササコア相当である Execution Unit とコア間の通信を担当する Communication Unit，投機的なストアデータの格納・管理を行う Memory Buffer，Memory Buffer の内容をメモリ上に反映する Write-Back Unit から構成される．

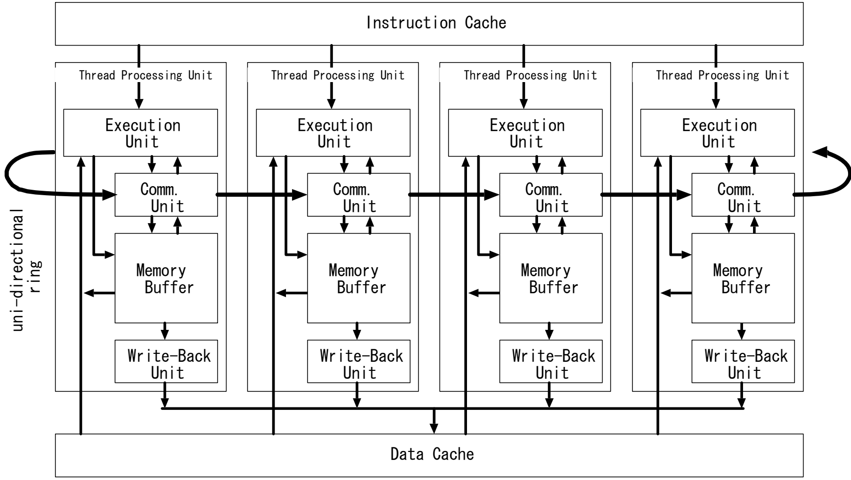

図 2・3 Superthreaded Architecture [4)]

本アーキテクチャでは，スレッドパイプライニングモデルと呼ばれる実行モデルに従った投機的マルチスレッディングが行われる．スレッドパイプライニングモデルでは，一つのスレッドは四つのステージに分けて実行される．最初のステージは Continuation stage と呼ばれ，次のスレッドの実行を開始するために必要な環境データ continuation を作成し，スレッド生成命令によって次のスレッドの実行を開始する．continuation の具体的な例として，ループの誘導変数の値などがあげられる．2 番目のステージは TSAG（Target-Store-Address-Generation）stage と呼ばれ，スレッド間でデータ依存となる可能性のあるメモリ上の変数のアドレスを計算して，Memory Buffer に対して登録する．Memory Buffer は登録されたアドレスについて，メモリアクセスの監視を行い，後続スレッドが登録されたアドレスに対してロード要求を出した際に，まだ対応するデータがストアが行われていない場合に待たせる役目を果たす．これによって，メモリデータについての同期通信が実現される．3 番目のステージは，Computation stage と呼ばれ，スレッドに割り当てられた本来の処理内容を実行する．この際，スレッド間で依存するメモリデータについて，そのスレッド内での最終値が確定した段階で，スレッド間データ転送用の命令コードによって，後続スレッドに Memory Buffer を通じて値が転送される．最後のステージは，Write-Back stage と呼ばれ，スレッドの実行結果をメモリ上に反映する．このステージはスレッド間でプログラム順に逐次化され，スレッド間での実行結果が元の逐次プログラムの実行結果と同じになるように同期処理される．

Superthreaded Architecture について特徴をまとめる．まず，スレッドの分割については Multiscalar と同様にコンパイラによって静的に行われる．Multiscalar とは異なり，機械命令を使ってスレッドの生成・終了などの制御を行う．各スレッドは，Multiscalar と同様に，リング接続されたプロセッサコア上で独立して実行され，スレッドの割り当て方もユニット間の接続形態により一意に決まっている．スレッド間でレジスタのイメージは非共有であり，データの受け渡しはロードストア命令によってのみ行われる．ただし，スレッド生成時点に

おいて親スレッドから子スレッドへとレジスタイメージのコピーが行われる．最後に，メモリデータについては，TSAG stage の処理とスレッド間データ転送命令を使ってソフトウェア制御によりスレッド間依存データの同期通信を行う．そのため，Multiscalar とは異なり，本アーキテクチャではデータ投機実行を行わない．

2-3-5 Trace Processors

Trace Processors[7] は，トレースキャッシュに基づいて，プログラムの実行トレースを単位に投機的マルチスレッディングを行うことで，逐次シングルプログラムの高速化を図ったものである．スレッドの分割はトレースキャッシュ中の各トレースという形で自然に実現される．次にどのトレースが実行されるかを予測しながら，それぞれのトレースを別個のスレッドとして並列に実行する．Trace Processors におけるプログラムは逐次のシングルプログラムそのものであり，これを実行しながらハードウェアによって自動的に(トレースという形で)スレッドの分割及び投機的マルチスレッディングを行う．

図 2･4 に Trace Processors のアーキテクチャを示す．スレッドを実行する Processing Element（以下，PE）が並んだ構成をとる．PE は小規模なスーパースカラプロセッサコアである．PE は，スレッド間でイメージを共有するグローバルなレジスタファイル，スレッドの実行内でローカルに使用するレジスタファイルをもつ．トレース予測器によって次に実行されるトレースを予測し，PE に割り当てる．また，そのトレースの実行に使用されるレジスタデータについて値予測器を用いて予測を行い，トレースとともに予測した値を PE に送る．PE は，割り当てられたトレースを予測値を用いて投機実行を行う．Trace Processors の全体的な構成は，トレース単位にディスパッチされるスーパースカラプロセッサのように見えるかも知れない．

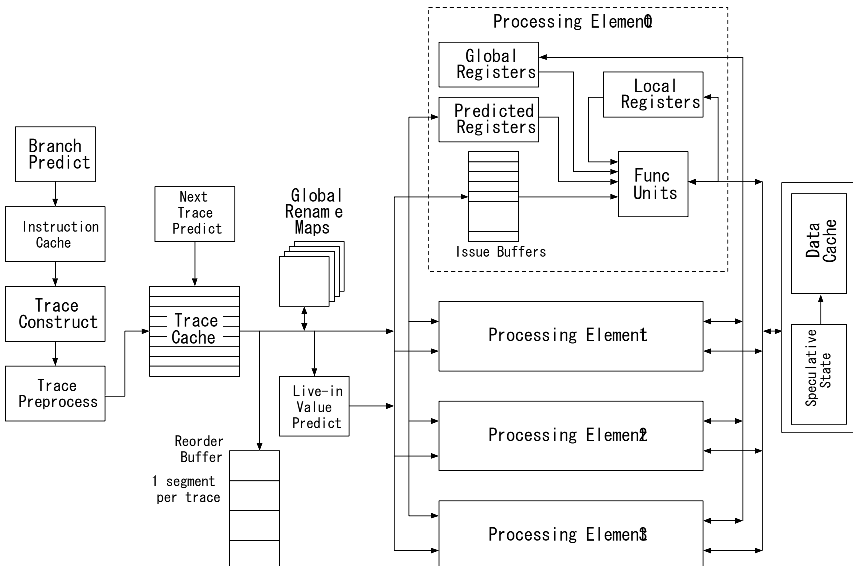

図 2・4　Trace Processors [7]

Trace Processors について特徴をまとめる．まず，スレッドの分割についてはハードウェアによって自動的に実行時に行われる．スレッドの割り当ては，プログラム順に空いているスレッド処理ユニットを使ってスレッドが実行される．各スレッド処理ユニットは独立したプロセッサコアである．スレッド間でレジスタのイメージを共有する方式をとっており，グローバルなレジスタファイルに最新の値が順次反映される．また，各スレッドはレジスタデータについての値予測に基づいてデータ投機実行を行う．メモリデータについては，Multiscalar と同様に，後続のスレッドは先行スレッドから送られてくるデータの到着を待たずにデータ投機実行を行う方式を採用している．

■参考文献

1) T. Ungerer, B. Robic, J. Silc, “Multithreaded Processors,” The Computer Journal, vol.45, no.3, pp.320-348, 2002.
2) M. Edahiro, S. Matsushita, M. Yamashina, and N. Nishi, “A Single-Chip Multiprocessor for Smart Terminals,” IEEE Micro, vol.20, no.4, pp.12-20, Jul./Aug. 2000.
3) G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, “Multiscalar Processors,” In Proc. of the 22nd Annual International Symposium on Computer Architecture, pp.414-425, 1995.
4) J. Tsai and P Yew, “The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation,” In Proc. of International Conference on Parallel Architectures and Compilation Techniques, pp.35-46, 1996.
5) 小林良太郎，小川行宏，岩田充晃，安藤秀樹，島田俊夫，“非数値計算応用向けスレッド・レベル並列処理マルチプロセッサ・アーキテクチャ SKY,” 情報処理学会論文誌, vol.42, no.2, pp.349-366, 2001.
6) 鳥居　淳，近藤真己，本村真人，池野晃久，小長谷明彦，西　直樹，“オンチップ制御並列プロセッサ MUSCAT の提案,” 情報処理学会論文誌, vol.39, no.6, pp.1622-1631, 1998.
7) E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, “Trace Processors,” In Proc. of the 30th Annual International Symposium on Microarchitecture, pp.138-148, 1997.
8) P. Marcuello, A. Gonzalez, and J. Tubella, “Speculative multithreaded processors,” In Proc. of International Conference on Supercomputing, pp.77-84, 1998.
9) H. Akkary and M. A. Driscoll, “A Dynamic Multithreading Processor,” In Proc. of the 31th Annual International Symposium on Microarchitecture, pp.226-236, 1998.
10) K. Hiraki, J. Tamatsukuri, and T. Matsumoto, “Speculative Execution Model with Duplication,” In Proc. of International Conference on Supercomputing, pp.321-328, 1998.
11) I. Park, B. Falsafi, and T. N. Vijaykumar, “Implicitly-Multithreaded Processors,” In Proc. of the 30th Annual International Symposium on Computer Architecture, pp.39-50, 2003.