1. アーキテクチャ基盤技術

2 スーパースカラ/VLIW プロセッサとスループット指向 MT プロセッサ

五島 正裕
東京大学 情報理工学系研究科
goshima@mtl.t.u-tokyo.ac.jp

スーパースカラ vs. VLIW プロセッサ

ここ十年ほどのマイクロアーキテクチャ研究の最大の成果の1つには，スーパースカラ vs. VLIW プロセッサの性能競争に決着を見たことが挙げられよう．

プログラムに内在する **ILP**（Instruction-Level Parallelism，命令レベル並列性）を抽出することによって性能の向上を図るプロセッサは，**ILP プロセッサ**と呼ばれる．ILP プロセッサは，通常，**スーパースカラ・プロセッサ**と **VLIW プロセッサ**の2つに大別される．

しかし，明暗を分けるのはスーパースカラか VLIW かにはない．今後の LSI 技術の方向性を考えると，out-of-order でなければ高い性能／効率は望めない．

■ In-Order/Out-of-Order

通常の（制御駆動型の）命令セット・アーキテクチャは，プログラム中の命令の逐次的な実行順序を定義する．この順序を，**プログラム・オーダ**と呼ぶ．

ILP プロセッサのうち，命令を必ずプログラム・オーダどおりに処理するものを **in-order プロセッサ**，プログラム・オーダどおりに処理しないことを許すものを **out-of-order プロセッサ**という．

In-order/out-of-order の違いが明確になるのは，何らかの理由で命令の実行が遅れた場合である．そのような場合，in-order プロセッサは，遅れた命令が実行されるまで後続の命令を実行しない．一方 out-of-order プロセッサは，その命令に依存しない後続の命令を先に実行することができる．**図-1** に示したパイプライン図では，最初のロード命令がキャッシュ・ミスを起こしている．In-order プロセッサでは，パイプライン・ストールにより，その命令に依存しない後続の命令の実行が丸ごと遅らされている．

In-order/out-of-order という分類は，通常スーパースカラ・プロセッサに対するものであるが，VLIW プロセッサにも敷衍することができる．VLIW プロセッサは，通常，ある長命令に含まれる命令の実行が遅れた場合，後続の長命令の実行を行わない．そのような意味で，VLIW プロセッサは in-order プロセッサであると考えることができる．

前述したように，明暗を分けるのは，スーパースカラか VLIW かではなく，in-order か out-of-order かにある．このことを理解するためにはまず，LSI の微細化の影響について考える必要がある．

LSI の微細化とマイクロアーキテクチャ

マイクロアーキテクチャは，その時々の LSI の性質に強く依存する．今後のマイクロアーキテクチャに最も重大な影響を与えるものの1つには，配線遅延がある．

■ LSI の微細化と配線遅延

半導体のプロセスが1世代進むと，最小加工寸法は約 $1/\sqrt{2}$ に縮小される．古典的なスケーリング則によれば，最小加工寸法が $1/S$ に縮小されると，ロジックの遅延も $1/S$ に短縮されるはずである．しかし，このような楽観的な法則は，現在の LSI にはもはや当てはまらない．ロジックの遅延は，**ゲート遅延**（gate delay）と**配線遅延**（wire delay）とに分けて考えることができる．

LSI の最小加工寸法が $1/S$ に縮小されると，ゲート遅延はほぼ $1/S$ に短縮される．前述した古典的なスケーリング則は，ゲート遅延のみを考慮したものである．

ゲート遅延に対して，配線遅延は微細化の恩恵を受けにくい．配線遅延は，分布 RC 回路の終端における遅延の1次の近似として，$1/2 \times RCL^2$ と表すことができる．

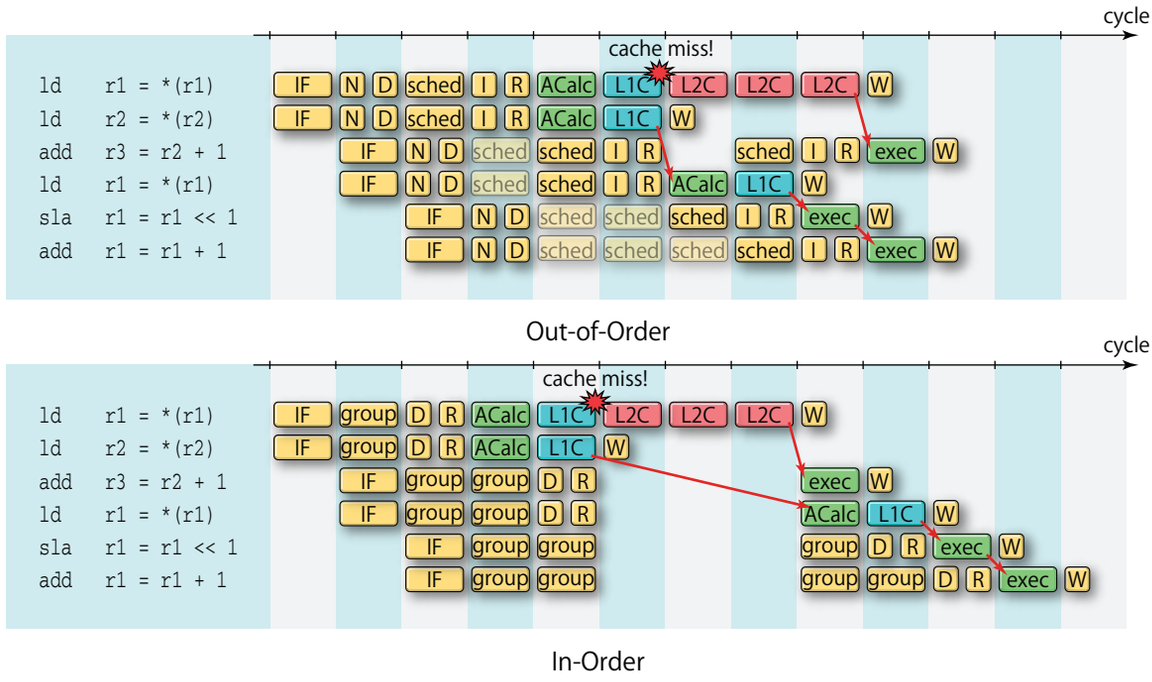

図-1　In-Order/Out-of-Order スーパースカラ・プロセッサのパイプライン図

R，C は，配線の単位長さあたりの抵抗と容量，L は配線の長さである．最小加工寸法が $1/S$ に縮小されるとき，回路の横方向の寸法がすべて $1/S$ に縮小されると考えよう．その場合，配線幅も同じく $1/S$ に縮小されるため，単位長さあたりの抵抗は S 倍になる．また，隣接する配線との距離も $1/S$ になるため，結合容量 C も S 倍になる．一方で，配線長 L も $1/S$ に短縮される．結局，最小加工寸法が $1/S$ に縮小されると，配線遅延 $1/2 \times RCL^2$ は $S \times S \times (1/S)^2 = 1$ 倍となる．すなわち，LSI の微細化に対して，配線遅延はまったく短縮されないのである．

実際には，すべての配線の長さが $1/S$ に短縮されるわけではないため，事態はより深刻である．クロック線や，キャッシュのワード線，ビット線などは，チップ・サイズの増加にしたがってむしろ長くなる傾向にある．それらの配線の遅延は，LSI が微細化されるにつれむしろ増大することになる．

配線遅延が短縮されにくい一方で，ゲート遅延は比較的順調に短縮されている．このことは，LSI が微細化されるにつれ，ロジックの遅延に占める配線遅延の割合が次第に増加していき，最終的にはロジックの遅延全体を支配するようになることを示している．

実際，近年，配線遅延の影響が顕在化しつつある．数世代前までは，ロジックの遅延はゲート段数によって近似することができた．ところが近年では，配線遅延の影響を考慮せずにロジックの遅延を見積もることはほとんど無意味になっている．

そのため，銅配線や，低誘電率層間絶縁膜──いわゆる，low-k 膜──など，配線遅延を短縮する技術の採用，開発が進んでいる．しかしこれらの技術は，RC に関する定数を改善するものであるので，問題をそれぞれ 1 世代分程度先延ばしにするにすぎない．

■ LSI の微細化がアーキテクチャに与える影響

ILP プロセッサを構成する種々のユニットは，LSI の微細化に関連して，演算器とそれ以外のユニットとに分けることができる．演算器とは，ALU，シフタ，アドレス計算器，浮動小数演算器などである．演算器の遅延は主にゲート遅延からなり，それ以外のユニットの遅延は配線遅延を多く含む．したがって，LSI が微細化されると，演算器の遅延は順調に短縮されるのに対し，演算器以外のユニットの遅延はあまり短縮されないことになる．

後述するように，キャッシュなどの一部のユニットでは，すでにその遅延が ALU のそれより長くなってしまっている．もしマイクロアーキテクチャや論理設計は変更せず，LSI が微細化されるにまかせていけば，いずれこれらのユニットがクリティカルになる，すなわち，これらのユニットの遅延がシステム全体のクロック速度を制限することになる．配線遅延はほとんど短縮されないため，それらの遅延が実際にクリティカルになれば，クロック速度もほとんど高速化されなくなってしまう．

■マイクロアーキテクチャ的対処

LSI の微細化に関するこのような傾向は，非集中化（decentralized）されたマイクロアーキテクチャの採用を促す．あるユニットがクリティカルでなくなるように，そのユニットの 1 クロック・サイクルあたりの遅延を減

らすには，つまるところ，そのユニットに費やすサイクル数を増やすか，ユニットの遅延自体を減らすしかない．サイクル数を増やすことはパイプライン化によって，遅延を減らすことは可変レイテンシ化によって，それぞれ達成できる．どちらの技術も，プロセッサを多くのより小さなユニットへと非集中化することになる．

可変レイテンシ化

可変レイテンシ化とは，縮小したユニットを用意し，元のユニットと組み合わせる手法である．縮小したユニットは，小型ゆえ低遅延であるが，一部の入力に対してしか所望の結果が得られない．通常は縮小したユニットを使用する（ヒット）が，所望の結果を得られなかった場合には，元のユニットを使用して，より多いサイクル数をかけて所望の結果を得ることになる（ミス）．可変レイテンシ化を施すと，ヒット／ミスに応じて，レイテンシが動的に変化することになる．

可変レイテンシ化の例としては，実際に演算器を可変レイテンシ化するほか，キャッシュの階層化や，演算器のクラスタ化などが挙げられる．本稿の文脈では，値予測などもこの範疇に含められる．

■キャッシュに対するパイプライン化と階層化

Intel Pentium 4 プロセッサや，AMD Athlon プロセッサなど，2 ～ 3GHz のクロック速度を持つプロセッサの 1 次データ・キャッシュは，パイプライン化／可変レイテンシ化の好例となっている．

データ・キャッシュのパイプライン化

これらのプロセッサの 1 次データ・キャッシュはすでにパイプライン化されており，レイテンシは 2（ないし 3）サイクルとなっている．ロード命令のレイテンシは，アドレス計算 1 サイクルと合わせて，3（ないし 4）サイクルになる．このことは，クロック速度が数 GHz に達すると，1（ないし 2）サイクルで十数～数十 KB 程度のキャッシュにアクセスすることが困難であることを示している．

データ・キャッシュをパイプライン化すると，主にロード命令のレイテンシが増加することによって，IPC はかなり低下する．しかし，クロック速度の向上の効果がそれを上回るため，より高い性能を得ることができるのである．

データ・キャッシュの階層化

ただし，ロード命令のレイテンシが 4 サイクルにもなると，さすがに IPC の低下の影響が深刻になる．そこで，1 種の 0 次キャッシュを導入することが提案されている[1]．

この 0 次キャッシュは，1 サイクルでアクセス可能になるように，その容量は 1KB 程度に制限されるが，70 ～ 80% 程度と，予想外に高いヒット率が得られる．そのため，ロード命令のレイテンシが 4 サイクルになった場合に，2 割程度の IPC の向上を達成することができる．

In-Order vs. Out-of-Order

以上の議論を踏まえて，in-order/out-of-order の性能／効率について論じよう．

以前には，in-order である VLIW プロセッサのほうが，（out-of-order）スーパースカラ・プロセッサより高い性能を発揮できると考えられていた．それは，out-of-order プロセッサの動的命令スケジューリング・ロジックが「複雑」で，クロック速度向上の足枷となると考えられていたためである．

■動的命令スケジューリング・ロジックの遅延

動的命令スケジューリングは，wakeup と select の 2 つのフェーズからなる．Out-of-order スーパースカラ・プロセッサでは，フェッチされた命令はすべていったん命令ウィンドウに格納される．格納された命令は，実行に必要なソース・オペランドが生成されるのを待って，命令ウィンドウ中で「眠る」ことになる．依存元の命令が実行され，ソース・オペランドが生成されると，その命令は「起こされ」，実行可能になる．Wakeup は，ソース・オペランドが生成されるのを追跡し，実行可能になる命令を見つけるフェーズである．Select は，wakeup によって見つけられた実行可能な命令の中から，実際に実行する命令を選択するフェーズである．

従来方式の Wakeup ロジック

Out-of-order スーパースカラ・プロセッサでは，プロセッサ内に存在するデータを一意に特定するため，すべてのデータにタグと呼ぶ ID が動的に割り当てられる．従来の wakeup ロジックは，このタグをキーとする連想検索に基づいている．命令が実行されると，その命令のデスティネーションに割り当てられたタグを命令ウィンドウ中で「眠っている」命令すべてに対して放送する．「眠っている」命令は，自らが待っているソースのタグと，放送されたタグとを比較する．タグが一致すれば，待っているソースが生成されたことが分かる．

従来の wakeup ロジックは，タグをキーとする連想メモリ── CAM（Content-Addressable Memory）を用いて実装される．**図 -2** に，従来方式の wakeup ロジックのブロック図を示す．同図では，0 番目の命令が実行されて，そのデスティネーションのタグ “10” が下部の CAM に入力されている．1，2 番の命令の左ソース・オペランドのタグがそれと一致するので，それらのオペランドが 利用可能になったと分かる．

この CAM は，RAM と比べると面積が大きく，その

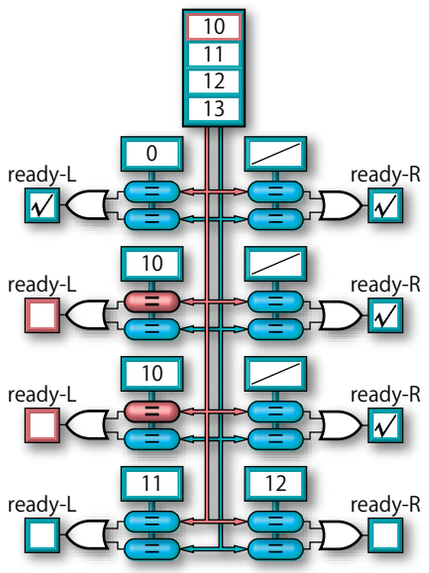

図-2　従来方式のWakeupロジック

遅延は配線遅延に支配される．そのため，LSI の微細化にともなってこのロジックの遅延がクリティカルになると考えられていた．

依存行列に基づく Wakeup ロジック

しかし近年では，低遅延な wakeup ロジックが開発されている[2), 3)]．この方式では，CAM ではなく，命令間の依存関係を表す依存行列を用いる．

図 -3 に，依存行列の概念図を示す．依存行列の行／列は命令ウィンドウ内の命令に対応する．命令ウィンドウの p 番エントリの命令の結果を，同じく c 番エントリの命令が使用する場合，c 行 p 列の要素を 1，それ以外の要素を 0 とする．

この行列を用いて wakeup の処理を行うには，p 番の命令が実行されるとき，単にこの行列の p 列を読み出せばよい．図 -3 の例では，図 -2 の場合と同様に，0 番の命令が実行されるときの様子を表している．この場合，0 行を読み出して，1，2 番の命令が実行可能になる．

一方，行列の更新は，命令間の依存関係を計算することであり，レジスタ・リネーミングと等価である．そのため，レジスタ・リネーミングと同時に行うことができ，更新のために性能が低下するということはない．

従来方式では，この依存関係の計算をも wakeup 時に行っていると考えることができる．依存行列を用いた手法では，その計算をレジスタ・リネーミング時に済ませておくことで，wakeup ロジックの簡単化を図っているのである．

さらに，依存行列の重要な成分のみを小容量のメモリにキャッシュする手法もあわせて提案されている．

このメモリは，遅延の観点からは，4bits × 4words

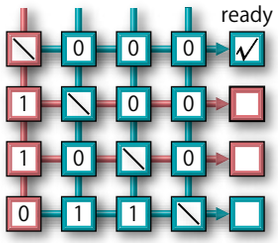

図-3　依存行列に基づくWakeupロジック

程度の RAM と等価となる．評価によれば，このロジックの遅延は F.O.4 インバータ 2 個分程度であり，このロジックがクリティカルになる可能性はきわめて低い．たとえば，SONY CELL プロセッサのサイクル・タイムは，F.O.4 インバータ 11 個分と報告されており[3)]，通常のプロセッサのサイクル・タイムは，それより長い．また，この程度の容量ならば，配線長も絶対的に短く，配線遅延の影響は将来にわたって小さくなる．

この方式などによって，動的命令スケジューリング・ロジックの遅延がクリティカルにはなる可能性はきわめて低くなった．VLIW プロセッサを含む in-order プロセッサのクロック速度上の優位性は失われたと言える．

■ In-Order プロセッサの効率の悪化

一方で，LSI の微細化にともなう配線遅延の増大のためにマイクロアーキテクチャが非集中化されていくにつれ，in-order プロセッサの効率の悪化が深刻化する．このことは，前述したデータ・キャッシュの例に顕著に見ることができる．

データ・キャッシュのパイプライン化

Out-of-order スーパースカラ・プロセッサの場合，キャッシュ・アクセスがパイプライン化され，ロード命令のレイテンシが増加した場合でも，そのロード命令に依存しない命令を見つけ，先に実行することができる．実際，先に例として挙げた Intel Pentium 4 プロセッサや AMD Athlon プロセッサなどは，out-of-order スーパースカラ・プロセッサである．

一方，in-order プロセッサの場合，その定義から言って，そのような対処は不可能である．ロード命令がミスを起こした時には，そのロード命令に依存する命令以降の命令の実行を丸ごと遅延させるしかない．

それでも以前は，このことはあまり問題にならなかった．1 次キャッシュのヒット率が 95% 程度以上と，十分高いためである．

しかし，キャッシュ・アクセスをパイプライン化すると，既存のバイナリの性能が著しく低下するという問題が発生する．既存のバイナリは，ロード命令のレイテン

シが2サイクル（アドレス計算：1，キャッシュ・アクセス：1）であると仮定してスケジューリングされている．そのため，キャッシュ・アクセスをパイプライン化すると，ほとんどロード命令を実行するたびごとにパイプライン・ストールが発生することになる．このようなバイナリは，コンパイルし直さなければとても使い物にならない．このことは，「性能上のバイナリ互換性の問題」と言ってよい．

データ・キャッシュの階層化

In-order プロセッサの場合，0次キャッシュ導入の効果も限定的である．

0次キャッシュのヒット率の70～80%は，予想外に高いとはいえ，1次キャッシュ・ヒット率とは比べるべくもない．Out-of-order プロセッサの場合，0次キャッシュ・ミス―1次キャッシュのレイテンシの大部分を，動的命令スケジューリングによって隠蔽することができる．一方，in-order プロセッサの場合には，0次キャッシュ・ミスのたびごとに1次キャッシュのレイテンシ分のストールが発生することになる．それによるIPCの低下は，2～4割にも及ぶ[3]．

■効率と省電力

このIPCの悪化は，省電力性能にも直接影響する．

直感的には，in-order プロセッサは，命令スケジューリング・ロジックの分だけout-of-order プロセッサより省電力であると考えがちだが，実際にはそうではない．In-order/out-of-order，2つのプロセッサで，同じ時間で同じプログラムを実行することを考えよう．In-order プロセッサはout-of-order プロセッサより2～4割IPCが低いとすると，同じ時間で実行するには，2～4割クロックを高速にする必要がある．消費電力量は，クロック速度の2～3乗に比例するから，1.4～2.7倍にもなる．動的命令スケジューリング・ロジックが，残りの部分の4割もの電力を消費するとは到底考えられない．

■ In-Order vs. Out-of-Order の結論

前述したように，out-of-order スーパースカラ・プロセッサの動的命令スケジューリング・ロジックの遅延はクリティカルになる可能性はきわめて低く，in-order プロセッサのクロック速度上の優位性は失われた．

一方で，LSIの微細化にともなう配線遅延の影響の増大のためアーキテクチャが非集中化されるにつれ，in-order プロセッサのIPCの悪化は深刻になっていく．

したがって，性能，および，効率の点で，VLIWを含むin-order プロセッサより，out-of-order プロセッサのほうが優れていると結論付けられる．

スループット指向 MT プロセッサ

ILP プロセッサは，同時に動作可能な複数の計算資源を持っている．しかし，当然のことではあるが，それらの計算資源のすべてを使いきれるわけではない．命令間の依存関係のため，プログラムに内在するILPには限りがある．また，分岐予測ミスやキャッシュ・ミスなどが発生した場合には，ミス処理の間プロセッサは半ば停止した状態にある．たとえば4命令を同時に実行できるILP プロセッサを用いても，実際にはIPCはせいぜい2程度しか出ないことが多い．その場合，計算資源は半分程度しか使われていないことになる．

そこで，単独ではIPCが2であるようなスレッドを2つ，1個のプロセッサで同時に実行することを考えよう．この場合，一方のスレッドで使うことができない計算資源をもう一方のスレッドが使うことによって，計算資源が有効に活用されると期待できる．最も理想的な場合には，本来1つのスレッドを実行する時間で2つのスレッドの実行を終えることができ，2倍のスループットを得ることができる．

このような発想を実際に実現しようとするのが，スループット指向のマルチスレッディング・プロセッサである．マルチスレッディングは，以下MTと略す．

■スループット指向 MT プロセッサの構成

MT プロセッサは，スレッドのコンテクストを複数保持する．スレッドのコンテクストとは，主に，レジスタとPCからなる．

MT プロセッサは，自分が保持するコンテクストを自分で切り替える．実装によっては，ソフトウェアに対して，1個のMT プロセッサをスレッド数個の非MT プロセッサに見せることもできる．

従来からあるMT プロセッサは，実際にあるタイミングでスレッドを切り替える．細粒度MTでは，サイクルごとに，粗粒度MTでは，キャッシュ・ミスなど，長い間プロセッサをストールさせるようなイベントの発生を契機として，スレッドを切り替える．

粗粒度MTは，SoEMT (Switch on Event MT) とも呼ばれ，最近再び注目を集めている．

■ SMT

SMT（Simultaneous Multi-Threading）[4] プロセッサは，simultaneous の名が示すとおり，あるタイミング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．**図-4** に，SMT プロセッサのあるサイクルのスナップショットを示す．同図に示され

ているように，SMT では，ある演算器ではスレッド 0 の，別の演算器ではスレッド 1 の命令がそれぞれ実行されているように見える．

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 MT よりさらに細粒度であると言える．

SMT プロセッサの構成

SMT は，既存の out-of-order スーパースカラ・プロセッサをベースに容易に MT 化することができる．Out-of-order スーパースカラ・プロセッサを SMT 化するには，スレッド・コンテク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．前述したように，out-of-order スーパースカラ・プロセッサはもともと，命令ウィンドウ中から実行可能な命令を見つけ（wakeup），空いている演算器に発行する（select）機能を持っている．したがって，ほとんど何の変更もなしに，複数スレッドから実行可能な命令を見つけ，演算器に発行することができる．データはタグによって一意に識別されるから，計算結果が混ざるというようなことはない．

SMT の利点

SMT は，従来の MT よりスレッドの粒度が小さいため，計算資源の利用率を高めやすい．特に粗粒度 MT では，スレッド切り替え時にパイプラインに別スレッドの命令が充填されるまでのオーバヘッドが発生し，分岐予測ミスによって生じる計算資源の空きを埋めることができない．細粒度 MT では，ミス時にパイプラインに充填されていた別スレッドの命令によって，SMT では，ミス時に命令ウィンドウに充填されていた別スレッドの命令によって，資源の空きを埋めることができる．

■スループット指向 MT プロセッサの採用例

SMT はその他の MT に比べて計算資源の利用率を高めやすいが，in-order プロセッサを SMT 化することはほとんど定義に反する．したがって，どの技術を採用するかは，ベースとなるプロセッサが in-order か out-of-order かによってほとんど決まってしまう．

Intel Pentium 4 プロセッサ[5] や，IBM POWER5 プロセッサは，out-of-order スーパースカラ・プロセッサをベースとしており，SMT を採用している．

一方，SONY CELL プロセッサの Power Processing Element[6]，Microsoft Xbox 360 CPU は，ともに in-order の PowerPC プロセッサをベースとしており，細粒度 MT を採用している．

また，Intel Montecito（次期 Itanium）や，Sun Niagara[7]（次期 SPARC）は，in-order プロセッサをベースとしており，粗粒度 MT を採用している．

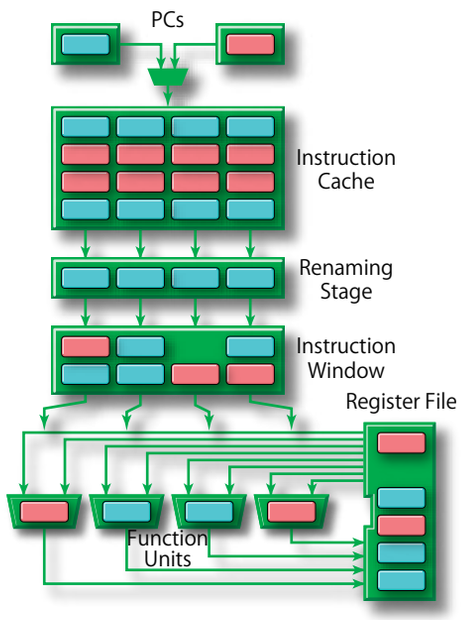

図-4　SMTプロセッサの概念図

■スループット指向 MT プロセッサの問題点

スループット指向 MT プロセッサには，以下のような問題点がある．

ワーキング・セット・サイズの増大

スループット指向 MT プロセッサでは，複数のスレッドを「同時に」実行するため，ワーキング・セット・サイズの増大が避けられない．

その結果として，まず，キャッシュ・ヒット率の低下が予想される．最悪の場合，スレッドあたりの実質的なキャッシュ容量がスレッド数分の 1 になってしまう．

レジスタ・ファイルの容量の増大

また，MT プロセッサでは，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．レジスタ数の増加は，上述したワーキング・セット・サイズの問題の一部であるが，最近の LSI ではキャッシュ以上に問題となる．

配線遅延の影響のため，キャッシュと同様，レジスタ・ファイルも非常に高価な資源となりつつある．レジスタ・ファイルは，1/2 サイクルでアクセスできることが理想であるが，すでに 1 サイクルでのアクセスも困難になっている．

そんな中，レジスタ数を増加させるには，クロック速度か IPC か，何らかのペナルティを払わなければならない．レジスタ数を増やすことは，前述したアーキテクチャの非集中化の傾向とは逆行する．

このため，スループット指向 MT プロセッサでは，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，厳しいトレードオフが発生することにな

る．このことは，ユーザにとっては，高クロック品がない，価格が高いといった状況として現れる．

ただし，電力効率が最重要視される超高密度サーバなどでは，電力効率のためにクロックを下げる結果として，このトレードオフは緩和される．

スケーラブルでない

スループット指向 MT プロセッサのポイントは計算資源の空きを埋めることにあるから，空いている以上にスレッド数を増やしても意味がない．IPC が同時実行命令数の 1/2 を大きく下回るとは考えにくいので，スレッド数は普通 2，多くとも 4 程度が限界である．現存する SMT プロセッサのスレッド数はすべて 2 となっている．

それ以上にスレッド数を増やすには，ベースとなる ILP プロセッサの同時実行命令数を増加させる必要があるが，これでは本末転倒である．そんなことをすれば，前述したレジスタの問題が一層深刻化することになる．

■スループット指向 MT プロセッサの用途

スループット指向 MT プロセッサでは，スレッド数とシングル・スレッドの性能／効率の間のトレードオフのため，使い方を選ぶ傾向が強い．

単一のプログラム

通常のクライアント PC のように，同時に実行したいプログラムが複数ない場合には，スループット指向 MT プロセッサを選択すべきではない．前述したように，サポートするスレッド数が多いほどシングル・スレッドの性能／効率は下がる．

複数のプログラム

サーバなど，定常的に複数のプログラムを実行する環境が，スループット指向 MT プロセッサのニッチである．Sun は，スループット・コンピューティングと称して，キャッシュ・ミスのレイテンシを MT によって隠蔽することを提唱している[7]．プリフェッチが有効でないプログラムに対しては，MT はキャッシュ・ミス・レイテンシを隠蔽するほとんど唯一有効な手段である．

ただし，非 MT プロセッサ環境に比べると，スレッドのスケジューリングは難しい．前述したように，むやみにスレッドをディスパッチしても，計算資源の空き以上に性能が向上することはなく，むしろワーキング・セット・サイズの増大によって性能が低下することも考えられる．

MT 化された単一のプログラム

単一のプログラムを MT 化してスループット指向 MT プロセッサで実行することも考えられなくはない．数値処理やメディア処理など，データ並列性を持つプログラムは比較的容易に MT 化できる．

しかしこれらのプログラムでは，もともと資源の利用率が高いため，マルチプロセッサやマルチコアを用いた場合のような大幅な IPC の向上は望めない．MT によって得られる IPC は，ソフトウェア･パイプライニング，キャッシュ･プリフェッチやタイリングなど，シングル･スレッド向けの最適化によっても達成できる可能性が高い．同程度の IPC が達成できるなら，スレッド数と性能／効率の間のトレードオフのため，非 MT プロセッサのほうが性能／効率が高い．

ただし，前述したように，電力効率が最重要視される超高密度サーバでは，トレードオフが緩和されるため，このような方向性が有効である可能性がある．

まとめと今後

本稿では，VLIW を含む in-order プロセッサは，性能／効率の点で，out-of-order スーパースカラ・プロセッサに敵わないと結論した．In-order プロセッサの明らかな優位点は，設計が容易なことくらいではないだろうか．少なくとも，in-order はロー・エンド向けの技術であると言える．

実際，配線遅延の影響にもかかわらず，out-of-order スーパースカラ・プロセッサの IPC は十年前の水準を維持している．それは，直接，間接に配線遅延に抗するさまざまな研究が続けられてきたためだ．配線遅延の問題に対しては，現在では一通り解決の方向性が示され，研究は最適化のフェーズに入ったように見える．今，out-of-order でなければならないと言えるのは，これらの研究成果があってこそである．

しかしその一方で，配線遅延など，デバイス技術が突きつけてくる問題点に対処するという研究のやり方自体に限界めいたものを感じずにはいられない．配線遅延に対しては目処が立った．次は，消費電力の問題か．では，その次の問題は？

そろそろこのようなやり方に決別するときが来たのではないだろうか．そのためには，デバイス技術には依存しない，斬新なマイクロアーキテクチャが必要だろう．

参考文献

1）Wilson, K. et al.: Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors, ISCA, pp. 147-157 (1995).
2）Goshima, M. et al.: A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors, MICRO-34, pp. 225-236 (2001).
3）五島正裕 : Out-of-Order ILP プロセッサにおける命令スケジューリングの高速化の研究 , 博士論文 , 京都大学 (2004).
4）Tullsen, D. et al.: Simultaneous Multithreading: Maximizing On-Chip Parallelism, ISCA (1995).
5）Marr, D. et al.: Hyper-Threading Technology Architecture and Microarchitecture, Intel Technology Journal, Vol.3, Issue 1, pp.4-15 (2002).
6）Pham, D. et al.: The Design and Implementation of a First-Generation CELL Processor, ISSCC 2005 (2005).
7）Bryg, W.: Niagara: Sun's Radical Realization of Chip Multithreading, Spring Processor Forum (2005).

（平成 17 年 8 月 19 日受付）