ハイパー・スレッディング・テクノロジの
アーキテクチャとマイクロアーキテクチャ

Deborah T. Marr -- インテル社、デスクトップ製品事業本部
Frank Binns -- インテル社、デスクトップ製品事業本部
David L. Hill -- インテル社、デスクトップ製品事業本部
Glenn Hinton -- インテル社、デスクトップ製品事業本部
David A. Koufaty -- インテル社、デスクトップ製品事業本部
J. Alan Miller -- インテル社、デスクトップ製品事業本部
Michael Upton -- インテル社、デスクトップ製品事業本部、CPU アーキテクチャ担当

検索用キーワード：アーキテクチャ、マイクロアーキテクチャ、ハイパー・スレッディング・テクノロジ、SMT (Simultaneous Multi-Threading)、マルチプロセッサ

摘要

インテルのハイパー・スレッディング・テクノロジは、インテル・アーキテクチャに SMT (Simultaneous Multi-Threading) の概念を導入したものです。ハイパー・スレッディング・テクノロジでは 1つの物理プロセッサが 2つの論理プロセッサとして認識されます。2つの論理プロセッサは物理的な実行リソースの大部分を共有しますが、アーキテクチャ・ステートに関してはそれぞれ専用のものが用意されます。これによって、ソフトウェアまたはアーキテクチャのレベルでは、オペレーティング・システムやユーザ・プログラムはこれまでの物理的なマルチプロセッサとまったく同様に、論理プロセッサに対してプロセス (スレッド) のスケジューリングを行えるようになります。マイクロアーキテクチャの観点から見ると、ハイパー・スレッディング・テクノロジでは 2つの論理プロセッサからの命令がそれぞれ独立性を維持しながら共有された実行リソース上で同時に実行できるという特徴があります。

本稿ではハイパー・スレッディング・テクノロジのアーキテクチャについて解説するとともに、ハイパー・スレッディング・テクノロジの初の実装

製品となるインテル® Xeon™ プロセッサ・ファミリのマイクロアーキテクチャについて詳しく説明します。ハイパー・スレッディング・テクノロジは、インテルのエンタープライズ向け製品ラインアップを強化する上で画期的な一歩となるものですが、今後は幅広い市場分野向けの製品に採用されていく予定です。

はじめに

インターネットと電気通信の驚異的な成長は、コンピュータ・システムの高速化によって支えられています。したがって、今後もこうしたシステムではより高いレベルのプロセッサ・パフォーマンスが要求されることになります。ところが、もはや従来のプロセッサ設計手法だけに頼っていては、こうしたニーズに応えられなくなっているのが現状です。これまで、プロセッサのパフォーマンスを高めるために、スーパーパイプライン、分岐予測、スーパースケーラ実行、アウトオブオーダー実行、キャッシュなどさまざまなマイクロアーキテクチャ手法が開発されてきました。ただし、これらはいずれもマイクロプロセッサを複雑化させ、トランジスタ数や消費電力の増大を招きます。事実、トランジスタ数と消費電力はプロセッサのパフォーマンスを上回るペースで増大しています。そこで、プロセッサの設計においては、いかにトランジスタ数や消費電力を抑えながらパフォーマンスを向上できるかが大きな課題となっています。その 1つのソリューションとなるのが、インテルのハイパー・スレッディング・テクノロジです。

プロセッサのマイクロアーキテクチャ

従来のプロセッサ設計では、クロック速度の高速化、命令レベルの並列化 (ILP)、およびキャッシュに重点を置いたアプローチがとられてきました。高クロッ

ク化を実現するための手法の 1 つに、マイクロアーキテクチャのパイプラインを細分化する「スーパーパイプライン」があります。クロック周波数を上げると 1 秒あたりに実行できる命令数が増えるため、パフォーマンスは大幅に向上します。ただし、スーパーパイプライン・マイクロアーキテクチャではきわめて多くの命令が同時にパイプラインを流れるため、キャッシュ・ミスや割り込み、分岐予測ミスなど、パイプラインを乱すイベントへの対処が重要になってきます。

ILP とは、1 クロック・サイクルでより多くの命令を実行できるようにするための手法です。例えば、スーパースケーラ・プロセッサには複数の並列実行ユニットがあり、複数の命令を同時に実行することができます。このため、スーパースケーラ実行を行うと、1 クロック・サイクルで数個の命令が実行できます。しかし、実行ユニットの数を増やしても、シンプルなインオーダー実行を行っていては十分な効果が得られません。ここで課題になるのは、実行ユニットに対していかに十分な命令を供給するかという点です。これを解決する手法として、アウトオブオーダー実行があります。これは、数多くの命令を同時に評価して、プログラムの並び順ではなく、命令の依存関係のないものから順に実行ユニットに送るというものです。

DRAM メモリについては、アクセス速度がプロセッサの実行速度よりもはるかに低速であるという問題があります。メモリ・アクセスのレイテンシを低減する方法の 1 つに、高速なキャッシュをプロセッサの近くに配置するというものがあります。頻繁にアクセスするデータや命令をキャッシュに格納しておけば、メモリ・アクセスは高速化します。しかし、一般にキャッシュの容量を大きくするとアクセス速度が低下してしまうため、プロセッサのキャッシュには階層化設計を採用するのが普通です。つまり、プロセッサ・コアの近くにはコアとほぼ同レイテンシでアクセスできる小容量で高速のキャッシュを配置し、以降は段階的に容量 (およびレイテンシ) の大きいキャッシュ・メモリを用意して、頻繁にアクセスする命令やデータから順に上位のキャッシュに格納するという手法です。しかし、どれだけキャッシュを用意してもすべてのデータをキャッシュに格納することは不可能で、キャッシュ・ミスは必ず発生します。すると DRAM メモリへのアクセスが必要となり、プロセッサに対する命令の供給が止まってしまいます。この状態をキャッシュ・ミスによるプロセッサのストールと呼びます。

プロセッサの世代交代期にはパフォーマンス向上のための新たな手法が数多く導入されますが、一般にこれらの手法はきわめて複雑で、プロセッサのダイ・サイズや消費電力を大幅に増大させてしまいます。しかも、これらの手法によって確かにパフォーマンスは向上しますが、100% の効率で向上するということはありません。つまり、プロセッサの実行ユニットの数を 2 倍に増やしても、命令フローの並列化には限度があるため、プロセッサのパフォーマンスは単純に 2 倍にはなりません。同様に、クロック周波数を 2 倍にしても分岐予測ミスが起こるとプロセッサ・サイクルを浪費してしまうため、やはりプロセッサのパフォーマンスが 2 倍になることはありません。

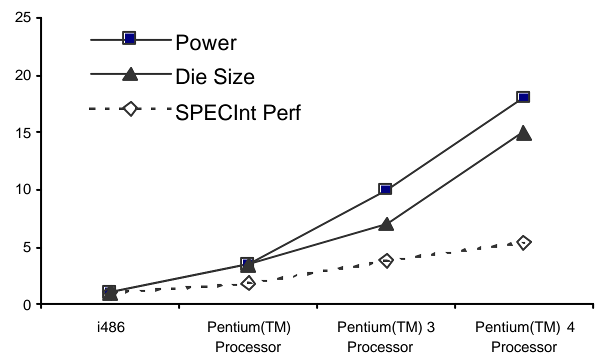

図 1：インテル® プロセッサの過去 10 年間のパフォーマンスとコストの推移

図 1 は過去 10 年間のインテル・プロセッサのパフォーマンスとコスト (ダイ・サイズと消費電力) の推移を示したものです[1]。純粋にマイクロアーキテクチャとしての比較を行うために、この表では 4 世代すべてのプロセッサを同一のシリコン・プロセス・テクノロジで製造したものと仮定し、パフォーマンスの比較は Intel486™ プロセッサのパフォーマンスを基準に相対的に行っています。なお、この表はインテル・プロセッサの過去 10 年間の推移ですが、他の高性能プロセッサ・メーカもほぼ同様の傾向をたどっています。表に示したとおり、インテル・プロセッサのパフォーマンスは、マイクロアーキテクチャの進歩だけでも整数演算性能 (SPECInt) が 5～6 倍に向上しています[1]。一般に、整数演算アプリケーションは ILP にも限界があり、しかも命令フローの予測が難しいという特徴があります。

一方、この間に相対的なダイ・サイズは 15 倍に達しており、これは整数演算性能の伸びの 3 倍に相当し

[1] このデータはおおよその値によって傾向を示すものであり、具体的なパフォーマンスを示すものではありません。

ます。ただし、シリコン・プロセス・テクノロジの進歩によってトランジスタの集積度も高まっている各世代のマイクロアーキテクチャの実際のダイ・サイズは、それほど大きくなっていません。

また、相対的な消費電力については、10年間で約18倍に増えています[1]。ただし、プロセッサの消費電力についても、省電力を実現するさまざまな技術が開発されており、現在でもこの分野ではさかんに研究が進められています。それでも現在のプロセッサの消費電力はデスクトップ・プラットフォームの許容範囲の限界に近づいているため、今後はプロセッサのパフォーマンスを高めるだけでなく、消費電力を抑えるための新しいテクノロジにも比重を置いた開発が強く求められます。

スレッド・レベルの並列化 (TLP)

現在のソフトウェアの傾向を見てみると、特にサーバ・アプリケーションではマルチスレッド化が進んでおり、複数のスレッド (プロセス) を並列に実行することができます。オンライン・トランザクション処理や Web サービスなどでは同時実行可能なスレッドが数多く存在するため、これらを並列に処理すればパフォーマンスが向上します。現在はデスクトップ・アプリケーションの分野でも並列化が進んでいます。インテルでは、このいわゆる「スレッド・レベルの並列化 (TLP)」を活用して、トランジスタ数や消費電力を最小限に抑えながらパフォーマンスを向上させようと考えています。

これまで、ハイエンドおよびミッドレンジのサーバ市場ではシステムのパフォーマンスを向上させるためにマルチプロセッサという手法が一般的に使われてきました。プロセッサの数を増やすと、複数のスレッドを複数のプロセッサで同時に実行できるため、アプリケーションのパフォーマンスが大幅に向上する可能性があります。ここでいう同時実行可能なスレッドには、同一アプリケーションのスレッド、同時に動作している複数のアプリケーションのスレッド、オペレーティング・システムのサービス、バックグラウンドでメンテナンスを行うオペレーティング・システムのスレッドなどが含まれます。マルチプロセッサ・システムが使われるようになってすでに長い年月が経っているため、ハイエンド・アプリケーションの開発現場ではマルチプロセッサの性能を存分に活用して優れたパフォーマンスを実現する手法が確立されています。

近年、TLP を利用する手法が数多く考案され、実際に製品化されたものもいくつかあります。その1つがチップ・マルチプロセシング (CMP) と呼ばれるものです。これは、1つのダイ上に2つのプロセッサを実装するというもので、実行リソースやアーキテクチャ・リソースもすべて2つずつ用意しておきます。大容量のオンチップ・キャッシュに関しては、2つのプロセッサで共有する場合もあります。CMP は従来のマルチプロセッサ・システムとは切り離して考えることができるため、CMP プロセッサをさらにマルチプロセッサ構成で使用することもできます。最近も、1つのダイに2プロセッサを実装した CMP プロセッサが発表されています。しかし、CMP チップは単一コアのチップに比べサイズが著しく大きく、製造コストが高くなるという問題があります。CMP はダイ・サイズや消費電力の面での取り組みがまだ始まっていません。

このほか、1つのプロセッサで複数のスレッドを切り替えて実行する方法もあります。タイムスライス・マルチスレッディングと呼ばれる方法では、プロセッサは一定時間ごとに複数のソフトウェア・スレッドを切り替えて実行します。タイムスライスのマルチスレッディングは実行スロットが有効に活用されないこともありますが、メモリ・アクセスのレイテンシの影響は効果的に最小化することができます。一方、スイッチ・オン・イベント・マルチスレッディングと呼ばれる方法では、キャッシュ・ミスなどレイテンシの大きいイベントが発生するとスレッドの切り替えを行います。このアプローチは、サーバ・アプリケーションのようにキャッシュ・ミスが大量に発生し、しかも2つのスレッドが類似したタスクを実行しているような場合には特に効果を発揮します。しかし、タイムスライスやスイッチ・オン・イベントによるマルチスレッディングでは、分岐予測ミスや命令の依存などによって利用効率の下がったリソースを別のスレッドで効果的に利用することが難しいという欠点があります。

そしてもう1つ、SMT (Simultaneous Multi-Threading) と呼ばれる手法もあります。これは、切り替えは行わずに複数のスレッドを1つのプロセッサ上で実行するというもので、複数のスレッドが同時に実行され、各種リソースをより効率的に利用できるという特徴があります。この手法ではプロセッサ・リソースをきわめて有効に利用できるため、トランジスタ数や消費電力を抑えながらパフォーマンスを向上させることが可能です。

ハイパー・スレッディング・テクノロジは、この SMT のアプローチをインテル・アーキテクチャに導入したものです。本稿ではハイパー・スレッディング・テクノロジのアーキテクチャ、および同テクノロジを初めて実装したインテル® Xeon™ プロセッサ・ファミリについて解説していきます。

ハイパー・スレッディング・テクノロジのアーキテクチャ

ハイパー・スレッディング・テクノロジとは、1つの物理プロセッサを複数の論理プロセッサとして認識させるものです [11、12]。原理的には、2つの論理プロセッサで物理的な実行リソースの大部分を共有させ、アーキテクチャ・ステートに関しては各論理プロセッサに対してそれぞれ専用のものを用意しています。これによって、ソフトウェアまたはアーキテクチャのレベルでは、オペレーティング・システムやユーザ・プログラムはこれまでの物理的なマルチプロセッサとまったく同様に、論理プロセッサに対してプロセス (スレッド) のスケジューリングを行うことができます。また、マイクロアーキテクチャの観点から見ると、ハイパー・スレッディング・テクノロジでは2つの論理プロセッサからの命令がそれぞれ独立性を維持しながら共有された実行リソース上で同時に実行できるという特徴があります。

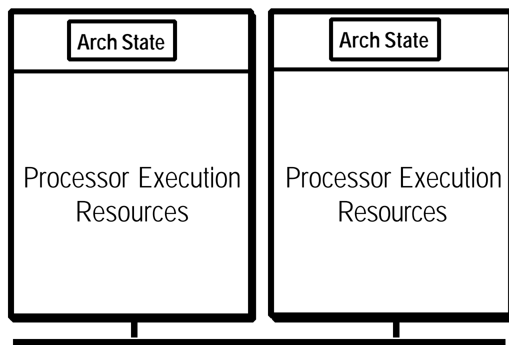

図 2：ハイパー・スレッディング・テクノロジに対応していないプロセッサ

図2は、ハイパー・スレッディング・テクノロジに対応していないプロセッサを物理的に2つ搭載したデュアル・プロセッサ・システムを例として示しています。一方、図3に示したのはハイパー・スレッディング・テクノロジに対応したプロセッサを物理的に2つ搭載したデュアル・プロセッサ・システムの模式図です。各物理プロセッサに2つのアーキテクチャ・ステートが用意されているため、このシステムには4つの論理プロセッサがあるものと認識されます。

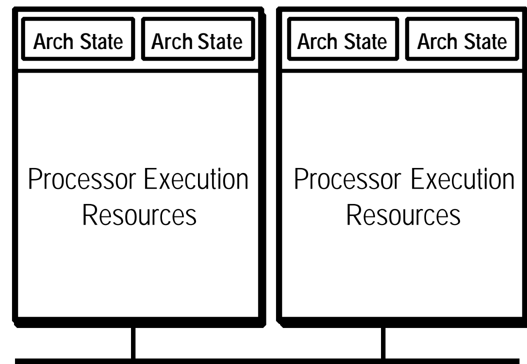

図 3：ハイパー・スレッディング・テクノロジに対応したプロセッサ

ハイパー・スレッディング・テクノロジを初めて実装した製品として、デュアル・プロセッサ/マルチプロセッサ・サーバ向けのインテル® Xeon™ プロセッサ・ファミリが登場しました。このプロセッサは、1つの物理プロセッサが2つの論理プロセッサとして認識されます。インテル Xeon プロセッサ・ファミリでは、既存のプロセッサ・リソースの利用効率を高めることによって、システムのコストは従来とほとんど同じレベルに抑えつつパフォーマンスを大幅に向上することに成功しています。事実、ハイパー・スレッディング・テクノロジの導入によって増加したチップ・サイズや最大消費電力は全体の5% 未満に抑えられているにもかかわらず、それをはるかに超えるパフォーマンス向上率を実現しています。

各論理プロセッサに対しては、専用のアーキテクチャ・ステート一式が用意されます。アーキテクチャ・ステートとは、汎用レジスタ、コントロール・レジスタ、APIC (Advanced Programmable Interrupt Controller) レジスタ、およびその他のマシン・ステート・レジスタで構成されます。このように2つのアーキテクチャ・ステートが存在するため、このプロセッサはソフトウェアから見た場合、2つのプロセッサとして認識されます。アーキテクチャ・ステートを二重化すると、当然トランジスタ数も増加しますが、これは全体的なトランジスタ数に比べるとほとんど問題にならない程度です。また、キャッシュ、実行ユニット、分岐予測器、コントロール・ロジック、バスなど、物理プロセッサ上のその他のリソースについては2つの論理プロセッサ間でほとんどすべてを共有します。

各論理プロセッサには専用の割り込みコントローラ (APIC) が用意されます。これによって、各論理プロセッサは、自分に送られた割り込みのみを処理できるようになっています。

インテル® XeonTM プロセッサ・ファミリにおける初の製品実装

ハイパー・スレッディング・テクノロジの初の実装製品となるインテル® XeonTM プロセッサ MP のマイクロアーキテクチャを設計する際には、いくつかの目標が設定されました。まず 1 つは、ダイ面積の増加を最小限に抑えながらハイパー・スレッディング・テクノロジを実装するという点です。この点については、2 つの論理プロセッサでマイクロアーキテクチャ・リソースの大部分を共有させ、二重化する部分をごく一部にとどめることで解決しました。事実、インテル Xeon プロセッサ MP ではハイパー・スレッディング・テクノロジの実装によって増加したダイ面積は全体の 5% 未満に抑えられています。

2 つめは、1 つの論理プロセッサがストールしても、もう一方の論理プロセッサが処理を続行できるようにすることです。キャッシュ・ミスや分岐予測ミス、あるいは前の命令の結果待ちなどが発生すると、論理プロセッサは一時的にストールします。そこで、2 つのアクティブなソフトウェア・スレッド [2] が実行中の場合は、どちらか一方の論理プロセッサがすべてのエントリを占有することがないよう、バッファリング・キューの設計に工夫を施しています。これによって、2 つの論理プロセッサが独立して処理を続行できるようにしています。具体的には、リソースを 2 つに分割 (パーティション化) したり、あるいは各スレッドが利用できるアクティブなエントリ数に制限を設けることによって実現しています。

3 つめは、アクティブなソフトウェア・スレッドが 1 つしかない場合、ハイパー・スレッディング・テクノロジ対応プロセッサの処理速度が通常のプロセッサの処理速度を下回らないようにするということです。このために、アクティブなソフトウェア・スレッドが 1 つしかない場合は、パーティション化されたリソースを再び結合して 1 つの論理プロセッサがすべてのリソースを使用できるようにしています。図 4 は、インテル Xeon プロセッサのマイクロアーキテクチャ・パイプラインを模式化したものです。ここに示したとおり、主なパイプライン・ロジック・ブロックの間にはバッファリング・キューが挿入されています。各バッファリング・キューはパーティション化 (1 つのものを半分に分割) または二重化 (同じものを 2 つ用意) されており、各ロジック・ブロックを通じて各スレッドが独立して処理を進行できるようになっています。

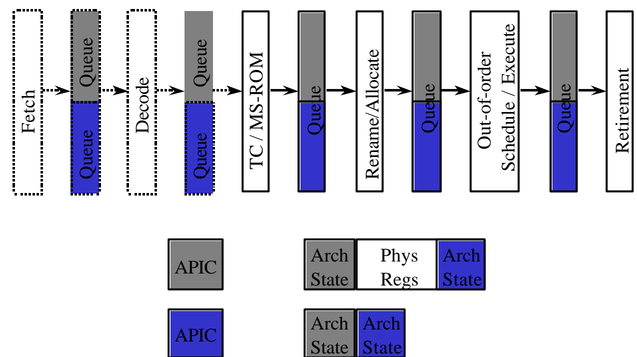

図 4：インテル® XeonTM プロセッサのパイプライン

以下のセクションでは、ハイパー・スレッディング・テクノロジにおける主な機能の実装方法やリソースの共有・二重化について、パイプラインの流れにしたがって解説します。

フロント・エンド

パイプラインのフロント・エンドの役割は、後のパイプ・ステージでの実行に備えて命令を供給することにあります。一般に、命令はレベル 1 (L1) 命令キャッシュに相当する「実行トレース・キャッシュ (TC)」からフェッチされます (図 5a)。また、必要な命令がトレース・キャッシュ内に見つからない場合は、プロセッサは統合レベル 2 (L2) キャッシュから命令をフェッチしてデコードを行います (図 5b)。トレース・キャッシュの近くにはマイクロコード ROM が配置されており、ここには IA-32 命令の中でも特にサイズの大きいものや複雑なものがデコードされた状態で格納されます。

[2] オペレーティング・システムのアイドル・ループもアクティブなソフトウェア・スレッドに含まれます。これは、アイドル・ループ時にもワーク・キューを常に監視するための一連のコードが実行されているためです。オペレーティング・システムのアイドル・ループも相当量の実行リソースを消費します。

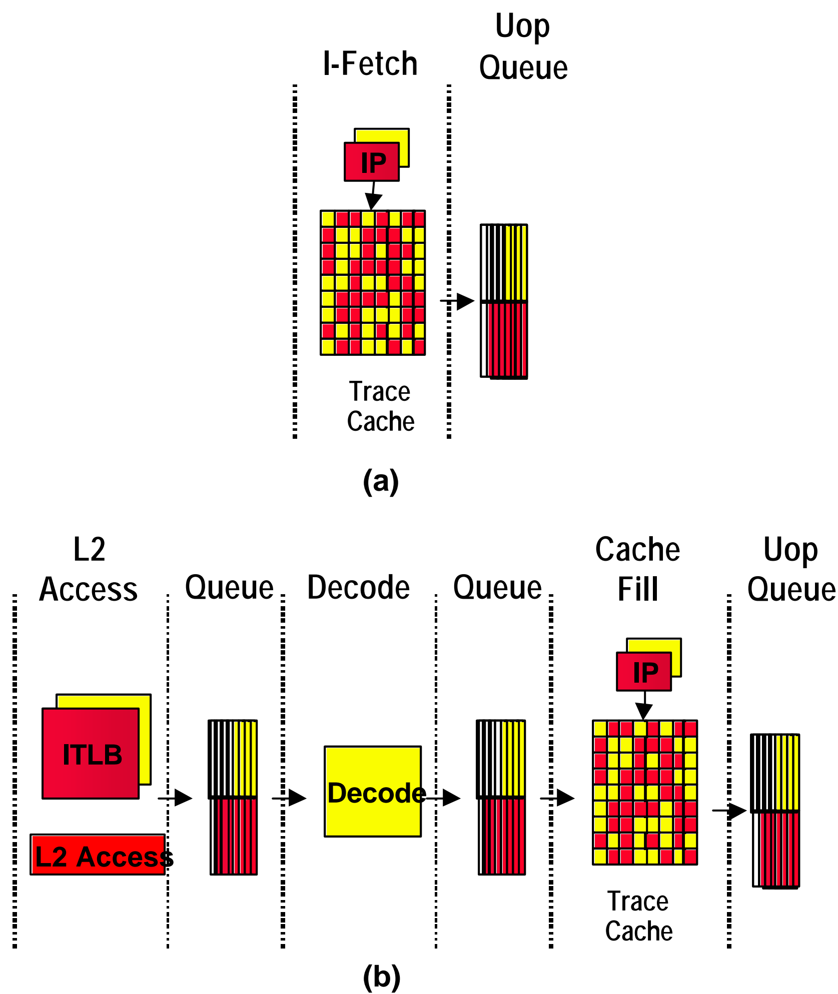

図 5：パイプラインのフロント・エンドの詳細図
(a) はトレース・キャッシュにヒットした場合、
(b) はトレース・キャッシュ・ミスの場合

実行トレース・キャッシュ

実行トレース・キャッシュには、μop (マイクロオペレーション) と呼ばれるデコード済みの命令が格納されます。ほとんどの場合、プログラム内の命令はこのトレース・キャッシュからフェッチして実行されます。命令ポインタは 2 つ用意されており、これらが独立して 2 つのソフトウェア・スレッドの実行状況を追跡できるようになっています。2 つの論理プロセッサは 1 クロック・サイクル単位でトレース・キャッシュへのアクセス権を交代します。両方の論理プロセッサが同時にトレース・キャッシュへのアクセスを要求すると、まず一方の論理プロセッサにアクセス権が与えられ、次のクロック・サイクルでもう一方の論理プロセッサにアクセス権が与えられます。例えば、1 クロック・サイクルで一方の論理プロセッサがラインのフェッチを行うと、次のサイクルではもう一方の論理プロセッサがラインのフェッチを行います。以上は両方の論理プロセッサが同時にトレース・キャッシュへのアクセスを要求した場合ですが、もし一方の論理プロセッサがストールしたり、あるいはトレース・キャッシュを利用できない状態にある場合は、もう一方の論理プロセッサが連続したクロック・サイクルでトレース・キャッシュにフルアクセスすることができます。

トレース・キャッシュのエントリにはスレッド情報のタグがつけられており、必要に応じて動的に割り当てが行われます。トレース・キャッシュは 8 ウェイ・セット・アソシアティブ・キャッシュで、エントリの置き換えはフル 8 ウェイをベースにした LRU (Least-Recently-Used) アルゴリズムに基づいて行われます。トレース・キャッシュは 2 つの論理プロセッサが共有

して使用するため、必要なら一方の論理プロセッサがもう一方の論理プロセッサよりも多くのエントリを使用することができます。

マイクロコード ROM

複雑な命令が現れると、トレース・キャッシュはマイクロコード ROM に対してマイクロコード命令ポインタを送ります。するとマイクロコード ROM コントローラが必要なμop をフェッチして、再び制御をトレース・キャッシュに返します。両方の論理プロセッサが同時に複雑な IA-32 命令を実行する場合に備え、マイクロコード命令ポインタも 2 つ用意されており、独立したフロー制御が可能になっています。

マイクロコード ROM のエントリも 2 つの論理プロセッサで共有します。また、1 クロック・サイクルごとに各論理プロセッサが交代でアクセスするといった動作も、トレース・キャッシュの場合と同様です。

ITLB および分岐予測

トレース・キャッシュ内に必要な命令が見つからない場合は、L2 キャッシュから命令バイトをフェッチして、μop にデコードしたものをトレース・キャッシュに格納します。この場合、ITLB (Instruction Translation Lookaside Buffer) はトレース・キャッシュから新しい命令を要求するリクエストを受け取ると、命令ポインタのアドレスを物理アドレスに変換します。そしてこのリクエストが L2 キャッシュに送られると、命令バイトが返されます。これらの命令バイトはストリーミング・バッファに格納されます。ストリーミング・バッファとは、デコード待ちの命令バイトを格納しておく場所です。

ハイパー・スレッディング・テクノロジでは ITLB が二重化されており、各論理プロセッサが 1 つずつ使用します。また、命令ポインタも論理プロセッサごとに用意されており、2 つの論理プロセッサに対する命令フェッチの進行状況を追跡できるようになっています。リクエストを L2 キャッシュに送る役目を果たす命令フェッチ・ロジックは、要求を受け取った順に処理していきますが、各論理プロセッサに対して少なくとも 1 つのリクエスト・スロットは常に確保するようにしています。こうすることで、両方の論理プロセッサは同時にフェッチ・ペンディングすることができます。

各論理プロセッサにはそれぞれ専用の 64 バイトのストリーミング・バッファが 2 つ用意されており、ここに命令デコード・ステージに送るまでの命令バイトを格納しておきます。ITLB やストリーミング・バッファはきわめてサイズが小さいため、これらを二重化してもダイ面積の増大はほとんど問題になりません。

分岐予測器には二重化されているものと共有されるものの 2 種類があります。リターン命令のターゲットを予測するリターン・スタック・バッファはサイズがきわめて小さいこと、そして呼び出し/リターンのペアの予測は各ソフトウェア・スレッドごとに独立して行った方が精度が高くなることなどから、各論理プロセッサごとに二重化しています。グローバル履歴アレイを参照するための分岐履歴バッファも各論理プロセッサごとに独立した追跡が行えるようになっています。ただし、グローバル履歴アレイに関してはサイズが大きいため、2 つの論理プロセッサが共有し、各エントリには論理プロセッサの ID タグがつけられるしくみになっています。

IA-32 命令デコード

IA-32 命令のデコードはやや複雑な手順を要します。これは、命令が可変長である上、オプションも命令ごとに異なるためです。したがって、IA-32 命令をデコードするには数多くのロジックや中間ステートが必要となります。ただし、ほとんどの場合トレース・キャッシュからμop をフェッチできるため、デコードが必要になるのはトレース・キャッシュ・ミスが発生した場合に限られます。

デコード・ロジックはストリーミング・バッファから命令バイトを取り出し、μop の形にデコードします。2 つのスレッドで同時に命令のデコードを行う場合は、ストリーミング・バッファは 2 つのスレッドを交互に切り替えることによって、両方のスレッドが 1 つのデコード・ロジックを共有できるようになっています。デコード・ロジックが一度に命令をデコードできるのは 1 つの論理プロセッサに対してのみですが、2 つの論理プロセッサに対して IA-32 命令のデコードを行うためにはステートを 2 つ用意しておく必要があります。通常、デコード・ロジックは一方の論理プロセッサに対して複数命令のデコードを行ってからもう一方の論理プロセッサに切り替えを行うようになっています。このように 2 つの論理プロセッサ間の切り替えを粗粒度で行うように設計したのは、ダイ・サイズおよび複雑さを抑えるためです。ここでも、一方の論理プロセッサのみがデコード・ロジックを必要としている場合は、その論理プロセッサがデコード・ロジックを完全に占有して使用することができます。デコードされた命令はトレース・キャッシュに書き込まれてからμop キューに送られます。

μop キュー

トレース・キャッシュまたはマイクロコード ROM からフェッチされたμop、あるいは命令デコード・ロジックから送られたμop は、「μop キュー」に格納されます。μop キューはパイプラインの流れにおい

てフロント・エンドとアウトオブオーダー実行エンジンの境界に位置するものです。μop キューはパーティション化されており、各論理プロセッサが全エントリの半分を使用することができます。μop キューを半分に分割することによって、フロント・エンドのストール(トレース・キャッシュ・ミスなど)や実行時のストールが発生しても、各論理プロセッサが独立して処理を進められるようになっています。

アウトオブオーダー実行エンジン

アウトオブオーダー実行エンジンは、割り当て、レジスタ・リネーム、スケジューリング、実行という一連の機能で構成されます(図 6)。アウトオブオーダー実行エンジンでは、命令の順序を並べ替えて実行、つまり本来のプログラム順には関係なく、入力オペランドが利用可能になった命令から順次実行していきます。

アロケータ

アウトオブオーダー実行エンジンには、命令のリオーダー、追跡、シーケンスを行うためのバッファがいくつか用意されており、アロケータ・ロジックがμop キューからμop を取り出して、各μop の実行に必要となる主なマシン・バッファ(126 エントリのリオーダー・バッファ、各 128 の整数および浮動小数点物理レジスタ、48 エントリのロード・バッファおよび 24 エントリのストア・バッファなど)の割り当ての大部分を行います。これらのマシン・バッファの中には 2 つの論理プロセッサのためにパーティション化されているものもあり、その場合は各論理プロセッサが最大でエントリの半分を使用するようになっています。具体的には、1 つの論理プロセッサが利用できるのは、リオーダー・バッファについては最大 63 エントリまで、ロード・バッファは 24 エントリ、ストア・バッファは 12 エントリまでとなります。

μop キュー内に両方の論理プロセッサに対するμop がある場合は、アロケータは 1 クロック・サイクルごとに論理プロセッサを切り替えながらμop の選択を行い、リソースの割り当てを行います。一方の論理プロセッサがストア・バッファのエントリなどの必要なリソースの割り当て量を使い切ってしまうと、アロケータはその論理プロセッサに対して「ストール」信号を送り、もう一方の論理プロセッサに対してリソースの割り当てを続行します。また、μop キュー内に一方の論理プロセッサに対するμop しか存在しない場合は、アロケータはすべてのクロック・サイクルでその論理プロセッサに対してリソースの割り当てを行うことで割り当て処理を高速に行えるようにしています。ただし、この場合も各論理プロセッサに対するリソースの割り当て制限がなくなるわけではありません。

主要バッファのリソース使用量に制限を設けることによって、各論理プロセッサ間の公平性を保ち、デッドロックを防ぐようにしています。

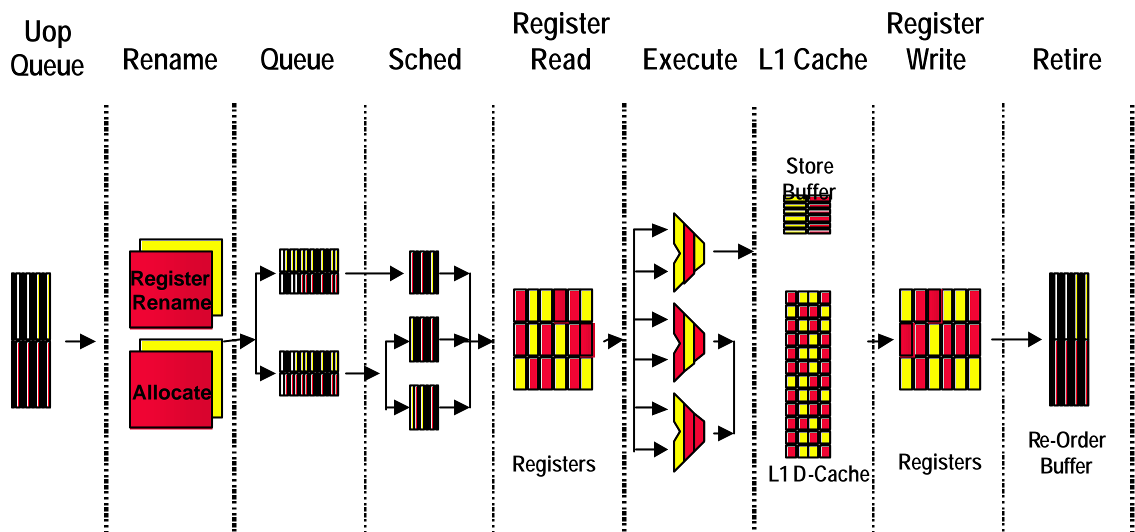

図 6：アウトオブオーダー実行エンジンのパイプラインの詳細

レジスタ・リネーム

レジスタ・リネーム・ロジックはIA-32アーキテクチャのレジスタをプロセッサの物理レジスタ上にリネームします。これにより、8エントリという制約のあるIA-32の汎用整数レジスタが動的に拡張され、プロセッサが備える128本の物理レジスタが使えるようになります。リネーム・ロジックではレジスタ・エイリアス・テーブル(RAT)を使って各アーキテクチャ・レジスタの最も新しい値を追跡し、次の命令に対する入力オペランドの場所を判断します。

ハイパー・スレッディング・テクノロジでは、各論理プロセッサはそれぞれ独自のアーキテクチャ・ステートを完全な形で維持、追跡しなければならないため、RATも各論理プロセッサに対して1つずつ専用のものが用意されています。レジスタ・リネームの処理は上記のアロケータ・ロジックと並行して行われるため、1つのμopに対してレジスタ・リネーム・ロジックによる処理とアロケータによるリソースの割り当て処理が同時に行われます。

μopに対するリソースの割り当てとレジスタ・リネームが完了したら、μopはメモリ操作(ロード/ストア)用キューとその他の汎用操作キューの2つのキューに格納されます。それぞれメモリ操作用キュー、汎用操作キューと呼ばれるこれらのキューはいずれもパーティション化されており、1つの論理プロセッサからのμopは最大でもエントリの半分までしか使えないようになっています。

命令スケジューリング

命令スケジューラはアウトオブオーダー実行エンジンの心臓部とも言えるものです。μopスケジューラは5つ用意されており、異なるタイプのμopをそれぞれ適切な実行ユニットに対してスケジューリングします。5つの命令スケジューラを合わせると、1クロック・サイクルで最大6つのμopをディスパッチできます。スケジューラは、依存関係にある入力レジスタ・オペランド・ソースが準備可能な状態にあるかどうか、およびそのμopの処理に必要な実行リソースが利用できるかどうかをもとに、μopがいつ実行可能になるかを判定します。

メモリ操作用キューと汎用操作キューは5つのスケジューラ・キューに対してなるべく高速にμopを送ります。2つの論理プロセッサのμopがある場合は、1クロック・サイクルごとに論理プロセッサを切り替えてμopを送ります。

各スケジューラにはそれぞれ8〜12エントリの専用スケジューラ・キューが用意されており、スケジューラはここからμopを選んで実行ユニットに送ります。スケジューラがμopを選ぶ際は、そのμopがどちらの論理プロセッサのものであるかは考慮しません。このように、スケジューラは論理プロセッサの区別を行いません。μopの評価は、依存関係にある入力、および実行リソースが利用可能であるかどうかのみを基準に行われます。例えば、スケジューラは一方の論理プロセッサからの2つのμopともう一方の論理プロセッサからの2つのμopを1クロック・サイクルで同時にディスパッチすることもできます。ただし、デッドロックを防ぎ、論理プロセッサ間の公平性を維持するため、1つの論理プロセッサがスケジューラ・キュー内に確保できるアクティブなエントリの数には制限が設けられています。この制限はスケジューラ・キューのサイズによって決まります。

実行ユニット

実行コアと階層型メモリも、論理プロセッサの区別をほとんど行いません。ソース・レジスタとデスティネーション・レジスタはすでに物理レジスタにリネームされて、物理レジスタ・プール内で共有されているため、μopは物理レジスタ・ファイルにアクセスすればデスティネーションが得られ、結果を物理レジスタ・ファイルに書き戻します。フォワーディング・ロジックは、物理レジスタ番号を比較すれば他の実行μopに結果を送れるため、特に論理プロセッサの区別を行う必要はありません。

実行が完了すると、μopはリオーダー・バッファに格納されます。リオーダー・バッファにより実行ステージとリタイアメント・ステージを分離しています。リオーダー・バッファはパーティション化されており、各論理プロセッサがエントリの半分を使用できるようになっています。

リタイアメント

μopリタイアメント・ロジックはアーキテクチャ・ステートをプログラム順にコミットします。リタイアメント・ロジックは2つの論理プロセッサからのμopのリタイア準備状況を追跡し、準備の整ったものから2つの論理プロセッサを交互に切り替えてμopをプログラム順にリタイアしていきます。つまり、リタイアメント・ロジックは一方の論理プロセッサのμopをリタイアしたら次はもう一方の論理プロセッサのμopをリタイアするという動作を繰り返します。もし片方の論理プロセッサにリタイア待ちのμopがまったくない場合は、連続したクロック・サイクルでもう一方の論理プロセッサのμopのリタイアが行われます。

ストアがリタイアすると、ストア・データを L1 データ・キャッシュに書き込む必要があります。2 つの論理プロセッサのストア・データを交代でキャッシュにコミットするために、セレクション・ロジックを使用しています。

メモリ・サブシステム

メモリ・サブシステムは、DTLB、低レイテンシのレベル 1 (L1) データ・キャッシュ、レベル 2 (L2) ユニファイド・キャッシュ、レベル 3 (L3) ユニファイド・キャッシュで構成されます (L3 キャッシュはインテル® Xeon™ プロセッサ MP のみ)。メモリ・サブシステムへのアクセスに関しても、論理プロセッサの区別はほとんど行いません。スケジューラは論理プロセッサの区別に関係なくロード/ストア μop を送り、メモリ・サブシステムはそれらを受け取った順に処理していきます。

DTLB

DTLB はアドレスを物理アドレスに変換します。DTLB には完全にアソシアティブな 64 のエントリが用意されており、各エントリは 4K または 4MB のページのいずれかをマッピングできます。DTLB は 2 つの論理プロセッサで共有されますが、各エントリには論理プロセッサの ID タグがつけられます。また、各論理プロセッサにはそれぞれ 1 つずつ予約レジスタも用意されており、公平性の維持、および DTLB ミス対応時の処理続行が行えるようになっています。

L1 データ・キャッシュ、L2 キャッシュ、L3 キャッシュ

L1 データ・キャッシュは 4 ウェイ・セット・アソシアティブ・キャッシュで、1 キャッシュ・ラインが 64 バイトの構成になっています。このキャッシュはライトスルー・キャッシュとなっており、L1 データ・キャッシュへの書き込み内容は同時に L2 キャッシュにも書き込まれます。L1 データ・キャッシュの管理は仮想アドレスと物理タグで行われます。

L2 キャッシュと L3 キャッシュは 8 ウェイ・セット・アソシアティブ・キャッシュで、1 キャッシュ・ラインが 128 バイトの構成になっています。L2 キャッシュと L3 キャッシュは物理アドレスで管理されます。なお、3 つの階層のキャッシュに格納されているエントリはすべて、どちらの論理プロセッサの μop が最初にそのデータをキャッシュに格納したかには関係なく、両方の論理プロセッサで共有できるようになっています。

2 つの論理プロセッサがキャッシュ内のデータを共有するため、キャッシュ内で競合が起こるとパフォーマンスが大きく低下する可能性もあります。しかし一方で、キャッシュ内のデータを 2 つの論理プロセッサが共有できる場合もあります。例えば、片方の論理プロセッサがプリフェッチしてキャッシュに格納した命令やデータを、もう一方の論理プロセッサが必要とする場合で、こうしたケースはサーバ・アプリケーションのコードではごく一般的です。供給者/消費者タイプの利用モデルの場合、一方の論理プロセッサが生成したデータをもう一方の論理プロセッサが使用するといったケースがよくあります。このような場合、パフォーマンスの大幅な向上が期待できます。

バス

論理プロセッサからメモリに対して送られたリクエストがどのキャッシュにもヒットしなかった場合は、バス・ロジックがリクエストを処理します。バス・ロジックにはローカル APIC 割り込みコントローラ、オフチップ・システム・メモリ、および I/O スペースが含まれます。また、バス・ロジックは、他の外部バス・エージェントが生成したリクエストのキャッシュ可能アドレス・コヒーレンシ (スヌープ) も処理します。さらに、ローカル APIC を経由した割り込みリクエストにも対応します。

サービスの観点から見ると、論理プロセッサからのリクエストは送られた順に処理され、キューやバッファなどの空間は共有されているように見えます。ここでは、どちらかの論理プロセッサに優先権が与えられるということはありません。

しかし、リクエストがどちらの論理プロセッサから送られたものであるかという情報は、バス・キューに格納されています。ローカル APIC および割り込みデリバリ・リソースに対するリクエストは、各論理プロセッサごとに固有のものです。また、バス・ロジックはバリア・フェンスおよびメモリの順序づけ操作の一部も実行します。この場合、操作は各論理プロセッサごとにバス・リクエスト・キューに対して行われます。

デバッグ処理、およびクラスタ化マルチプロセッサの実装における処理機構を支援するため、論理プロセッサの ID はトランザクションのリクエスト・フェーズにおいてプロセッサ外部バスに明示的に送られます。キャッシュ・ラインの解放要求やプリフェッチ・トランザクションなど、その他のバス・トランザクションについては、トランザクションを発生させたリクエストの論理プロセッサ ID を引き継ぎます。

シングルタスク・モードとマルチタスク・モード

実行すべきソフトウェア・スレッドが1つしかない場合でも十分なパフォーマンスが得られるよう、ハイパー・スレッディング・テクノロジにはシングルタスク (ST) モードとマルチタスク (MT) モードの2つの動作モードが用意されています。MT モードでは、2つの論理プロセッサがアクティブとなり、いくつかのリソースは前述のとおりパーティション化された状態となります。一方、ST モードには論理プロセッサ0のみがアクティブなシングルタスク・モード (ST0) と論理プロセッサ1のみがアクティブなシングルタスク・モード (ST1) があります。ST0 または ST1 モードでは、片方の論理プロセッサのみがアクティブであるため、MT モード時にパーティション化されていたリソースは結合され、そのリソースはアクティブな方の論理プロセッサがすべて使用できるようになります。IA-32 インテル・アーキテクチャには、HALT と呼ばれる命令があります。この命令は通常、プロセッサの実行を止め、プロセッサを低消費電力モードに移行させる働きをします。HALT は特権命令と呼ばれ、オペレーティング・システムまたはその他の ring-0 プロセスのみが実行権限を持っており、ユーザ・レベルのアプリケーションは HALT を実行できません。

ハイパー・スレッディング・テクノロジに対応したプロセッサでは、HALT 命令を実行するとプロセッサの状態が MT モードから ST0 または ST1 モードに移行します。ST0 モードになるか ST1 モードになるかは、どちらの論理プロセッサが HALT 命令を実行したかによって決まります。例えば、論理プロセッサ0が HALT 命令を実行すると、論理プロセッサ1のみがアクティブになります。このとき、物理プロセッサは ST1 モードに移行し、パーティション化されたリソースはすべて結合されて、論理プロセッサ1がすべてのプロセッサ・リソースを利用できるようになります。この状態で論理プロセッサ1も HALT 命令を実行すると、物理プロセッサは低消費電力モードに移行することができます。

HALT が実行されて ST0 モードまたは ST1 モードになったプロセッサに対して割り込みが送られると、プロセッサは MT モードへ移行します。このように、MT モードと ST モードの切り替えはオペレーティング・システムの役割となります (詳しくは次のセクションで解説)。

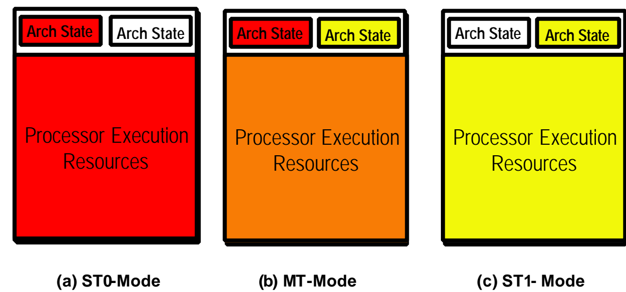

図 7：リソースの割り当て

以上の説明をまとめたものが図7です。ハイパー・スレッディング・テクノロジに対応したプロセッサでは、ST0 または ST1 モード時にはリソースはすべてどちらか一方のアクティブな論理プロセッサに割り当てられます。MT モード時には2つの論理プロセッサがリソースを共有します。

オペレーティング・システムとアプリケーション

ハイパー・スレッディング・テクノロジ対応プロセッサを搭載したシステムでは、オペレーティング・システムやアプリケーション・ソフトウェアは物理プロセッサ数の2倍のプロセッサを認識します。オペレーティング・システムが実行可能なタスクやスレッドをスケジューリングしていく動作については、物理的なマルチプロセッサの場合も論理プロセッサの場合も変わりありません。ただし、ハイパー・スレッディング・テクノロジの性能を十分に引き出すには、オペレーティング・システム側で2つの最適化を行う必要があります。

1つは、片方の論理プロセッサのみがアクティブになった場合には必ず HALT 命令を使用するということです。HALT 命令を実行すると、プロセッサは ST0 または ST1 モードのいずれかに移行します。オペレーティング・システム側でこの最適化を行わないと、アイドル状態になった論理プロセッサ上ではワーク・キューの状態を継続的にチェックするための一連のコードが実行されてしまいます。このいわゆる「アイドル・ループ」は相当量の実行リソースを消費するため、本来はアクティブな論理プロセッサが利用できるはずのリソースを占有してしまい、十分なパフォーマンスが得られなくなります。

もう1つは、論理プロセッサに対するソフトウェア・スレッドのスケジューリングの方法です。一般に、ハイパー・スレッディング・テクノロジで十分なパフォーマンスを実現するには、オペレーティング・システムがスレッドのスケジューリングを行う際に、まずは別々の物理プロセッサ上の論理プロセッサに対してスレッドのスケジューリングを行い、

その後で 1 つの物理プロセッサ上の複数の論理プロセッサに対してスレッドのスケジューリングを行うようにします。こうすることによって、なるべくソフトウェア・スレッドが別々の物理プロセッサの実行リソースを使用できるようにします。

パフォーマンス

インテル® Xeon™ プロセッサ・ファミリは、これまで発表された IA-32 インテル・アーキテクチャ・プロセッサとして最も優れたサーバ・システム・パフォーマンスを発揮します。初期のベンチマーク・テストでも、4 ウェイ・サーバ・プラットフォーム上でのハイエンド・サーバ・アプリケーションのパフォーマンスが従来の Pentium® III Xeon™ プロセッサに比べ、最大 65% 向上しています。この性能向上の主な要因は、ハイパー・スレッディング・テクノロジの導入にあります。

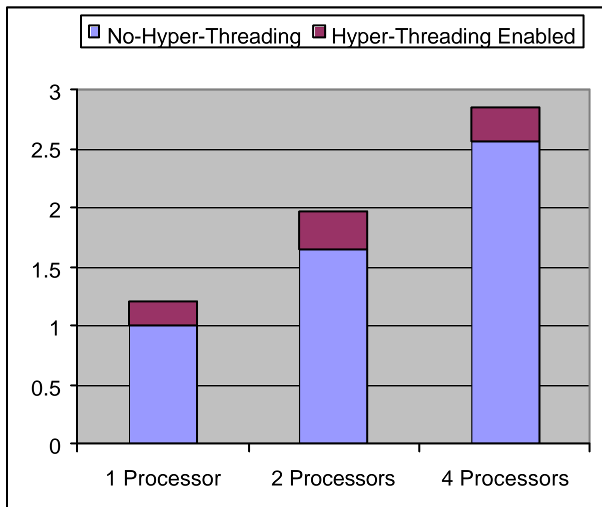

図 8：ハイパー・スレッディング・テクノロジがもたらす性能向上 (OLTP ワークロードの場合)

図 8 は OLTP (Online Transaction Processing) ワークロードに対するパフォーマンスを示したもので、左から順に 1 プロセッサ、2 プロセッサ、4 プロセッサ構成となっており、それぞれハイパー・スレッディング・テクノロジ有効時と無効時の性能をグラフにしています。ここでは、ハイパー・スレッディング・テクノロジ無効時の 1 プロセッサ構成のシステムのパフォーマンスを基準に比較しています。これを見ると、1 プロセッサおよび 2 プロセッサ・システムでは 21% の性能向上を示すなど、ハイパー・スレッディング・テクノロジを使用することによって全体的なパフォーマンスが大きく向上しているのが分かります。

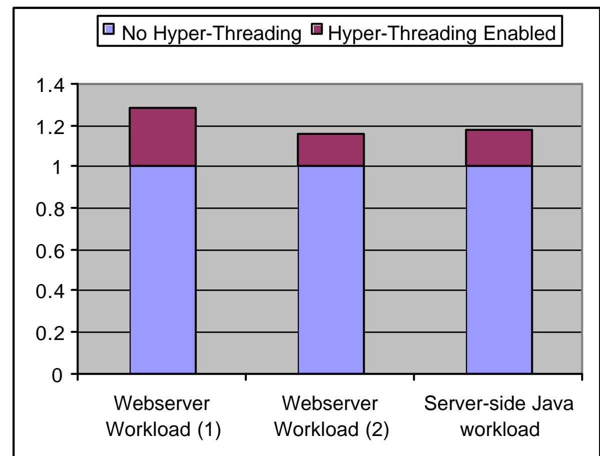

図 9：Web サーバのベンチマーク・パフォーマンス

図 9 は、その他のサーバ用ベンチマークにおけるハイパー・スレッディング・テクノロジの効果を示しています。ここでは、データ・サーバおよび Web サーバの性能を評価するための 2 種類のワークロードを用いたベンチマークと、サーバサイド Java アプリケーションの実行性能を評価するためのベンチマークを行っています。ここでも、ハイパー・スレッディング・テクノロジを利用することによって 16〜28% のパフォーマンス向上が認められます。

なお、上記のパフォーマンスの結果については絶対的なパフォーマンスではなく、相対的なパフォーマンスを示すことを目的に正規化を行っています。

パフォーマンス・テストや評価での測定値は、あくまでも特定のコンピュータ・システムやコンポーネントを使用した場合の数値であり、インテル製品のおおよそのパフォーマンスを示すものです。システム内のハードウェアまたはソフトウェアの設計/設定が異なると、実際のパフォーマンスにも差が生じます。システムやコンポーネントの購入を検討される場合は、ほかの情報も考慮に入れて、パフォーマンスを総合的に評価することをお勧めします。

まとめ

インテルのハイパー・スレッディング・テクノロジは、インテル・アーキテクチャに SMT (Simultaneous Multi-Threading) の概念を導入したものです。これはインテルの今後のプロセッサ・テクノロジの大きな方向性を示しています。ハイパー・スレッディング・テクノロジに関しては、今後さらにトランジスタ数と消費電力を抑えながらパフォーマンスの向上を実現する手法の開発が進むと考えられ

ますが、そうなるとハイパー・スレッディング・テクノロジの重要性は一層大きくなっていくでしょう。

インテル® Xeon™ プロセッサ MP は、ハイパー・スレッディング・テクノロジを初めて製品として実装したプロセッサです。インテル Xeon プロセッサ MP では、1つの物理プロセッサが2つの論理プロセッサとして認識されます。2つの論理プロセッサにはそれぞれ独立したアーキテクチャ・ステートが用意されていますが、プロセッサの実行リソースなど物理的なハードウェア・リソースはほとんどすべてを2つの論理プロセッサ間で共有します。インテル Xeon プロセッサ MP の開発に当たっては、コストの増加を最小限に抑えつつハイパー・スレッディング・テクノロジを実装すること、片方の論理プロセッサがストールしてももう一方が処理を続行できること、およびアクティブな論理プロセッサが1つしかない場合でも十分なパフォーマンスを発揮できるようにすること、という3つの目標が設定されました。インテル Xeon プロセッサ MP では、効率的な論理プロセッサ選択アルゴリズム、および主なリソースを動的にパーティション化/再結合させるアルゴリズムなどの採用により、この目標の達成に成功しています。

ハイパー・スレッディング・テクノロジを有効にしたインテル Xeon プロセッサ MP では、一般的なサーバ・アプリケーション・ベンチマークで最大30% の性能向上が認められます。

今回初めてインテル Xeon プロセッサ MP に実装されたハイパー・スレッディング・テクノロジは、今後の実装次第によってまだまだ大きな可能性が秘められています。ハイパー・スレッディング・テクノロジは今後モバイル・プロセッサからサーバ・プロセッサまで幅広い分野でその効果を発揮するものと考えられます。そのためには、サーバ以外の市場分野でもアプリケーションやワークロードのマルチスレッド化が進む必要があります。

謝辞

ハイパー・スレッディング・テクノロジは、設計者、検証スタッフ、アーキテクトなど多くの方々の甚大なる献身、プランニング、努力の成果として実現することができました。また、オペレーティング・システム・ベンダ、BIOS ベンダ、ソフトウェア・ベンダ各社による見事なチームワークもハイパー・スレッディング・テクノロジの実現に大きく貢献しました。特に、ハイパー・スレッディング・テクノロジの定義プロセスにおいて数々の助言をいただいたことを感謝します。現在も数多くのエンジニアが引き続きソフトウェア・ベンダ・パートナー各社と協力して、ハイパー・スレッディング・テクノロジにおけるアプリケーション・パフォーマンスの研究に従事しています。彼らのたゆまぬ努力と貢献は、ハイパー・スレッディング・テクノロジを画期的な技術として市場に投入する上で大きな役割を果たしています。

参考資料

[1] A. Agarwal、B.H. Lim、D. Kranz、J. Kubiatowicz 著『APRIL: A processor Architecture for Multiprocessing』、Proceedings of the 17th Annual International Symposium on Computer Architectures、1990年5月、pp. 104-114

[2] R. Alverson、D. Callahan、D. Cummings、B. Koblenz、A. Porter、B. Smith 著『The TERA Computer System』、International Conference on Supercomputing、1990年6月、pp. 1-6

[3] L. A. Barroso et. al. 著『Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing』、Proceedings of the 27th Annual International Symposium on Computer Architecture、2000年6月、pp. 282-293

[4] M. Fillo、S. Keckler、W. Dally、N. Carter、A. Chang、Y. Gurevich、W. Lee 著『The M-Machine Multicomputer』、28th Annual International Symposium on Microarchitecture、1995年11月

[5] L. Hammond、B. Nayfeh、K. Olukotun 著『A Single-Chip Multiprocessor』、Computer、1997年9月、pp. 79-85

[6] D. J. C. Johnson 著『HP's Mako Processor』、Microprocessor Forum、2001年10月、http://www.cpus.hp.com/technical_references/mpf_2001.pdf (英語)

[7] B.J. Smith 著『Architecture and Applications of the HEP Multiprocessor Computer System』、SPIE Real Time Signal Processing IV、1981年、pp. 241-248

[8] J. M. Tendler、S. Dodson、S. Fields 著『POWER4 System Microarchitecture』、Technical White Paper IBM Server Group、2001年10月

[9] D. Tullsen、S. Eggers、H. Levy 著『Simultaneous Multithreading: Maximizing On-chip Parallelism』、22nd Annual International Symposium on Computer Architecture、1995年6月

[10] D. Tullsen、S. Eggers、J. Emer、H. Levy、J. Lo、R. Stamm 著『Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading

processor』23rd Annual International Symposium on Computer Architecture、1996 年 5 月

[11]インテル コーポレーション、『IA-32 インテル® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、上巻：基本アーキテクチャ』注文番号 245470J、2001 年 http://www.intel.co.jp/jp/developer/design/pentium4/manuals/index.htm

[12]インテル コーポレーション、『IA-32 インテル® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、下巻：システム・プログラミング・ガイド』 注文番号 245472J、2001 年 http://www.intel.co.jp/jp/developer/design/pentium4/manuals/index.htm

著者紹介

Deborah T. Marr、デスクトップ製品事業本部、ハイパー・スレッディング・テクノロジ担当の CPU アーキテクト。インテル在籍 10年以上を数える。1988 年インテル入社以来、10 年以上にわたりインテル 386SX プロセッサ、P6 プロセッサ・マイクロアーキテクチャ、およびインテル® Pentium® 4 プロセッサ・マイクロアーキテクチャに多大な貢献を残す。高性能マイクロアーキテクチャおよびパフォーマンス解析を専門とする。1988年に カリフォルニア大学バークレー校にて電気工学/コンピュータ・サイエンス (EECS) の学士号、1992 年にコーネル大学にて電気工学/コンピュータ工学 (ECE) の修士号を取得。

電子メールアドレス：debbie.marr@intel.com

Frank Binns、サルフォード大学 (英国) にて電気工学の学士号を取得。Marconi Research Laboratories および Diamond Trading Company Research Laboratory (いずれも英国) にてリサーチ・エンジニアリングの職を経て、1984年インテル入社。インテルでの 17 年間、当初は開発ツール、マルチバス・システム、PC システムの各事業部でテクニカル・マネージャを務めた後、8 年前よりデスクトップ・プロセッサ事業本部 (テクニカル・マーケティング/プロセッサ・アーキテクチャ部門) に所属。

電子メールアドレス：frank.binns@intel.com

Dave L. Hill、1993 年にインテルのデスクトップ製品事業本部に配属以来、Pentium 4 プロセッサのバス・ロジック担当アーキテクトを務める。コンピュータ業界で 20 年間活躍しており、主に高性能メモリ・システムのマイクロアーキテクチャ、ロジック設計、システム・デバッグを専門とする。

電子メールアドレス：david.l.hill@intel.com

Glenn Hinton、インテル・フェロー、インテル・アーキテクチャ事業本部 IA-32 マイクロアーキテクチャ開発担当ディレクタ。1983 年にインテル入社。1990 年、Pentium® Pro、Pentium® II、Pentium® III、Celeron® プロセッサのベースとなる P6 マイクロアーキテクチャの開発に携わった 3 名の上級アーキテクトの 1 人。Pentium® 4 プロセッサのマイクロアーキテクチャ開発責任者を務める。1983 年、ブリガム・ヤング大学にて電気工学の修士号を取得。

電子メールアドレス：glenn.hinton@intel.com

David A. Koufaty、シモン・ボリバル大学 (ベネズエラ) にて 1988 年に理学士号、1991 年に理学修士号を取得。1997 年にはイリノイ大学アーバナ・シャンペーン校にてコンピュータ科学の博士号を取得。4 年前よりデスクトップ・プロセッサ事業本部の CPU アーキテクチャ部門に所属。マイクロプロセッサ・アーキテクチャおよびソフトウェア、パフォーマンス、コンパイルを専門とする。

電子メールアドレス：david.a.koufaty@intel.com

John (Alan) Miller、インテル在籍 5 年以上を数える。この間、Pentium 4 プロセッサおよび対応製品のプロジェクトの設計、アーキテクチャ面に携わる。カーネギー・メロン大学にて電気工学/コンピュータ工学 (ECE) の修士号を取得。

電子メールアドレス：alan.miller@intel.com

Michael Upton、インテル社、デスクトップ・プラットフォーム事業本部、主席エンジニア/アーキテクト。インテル® Pentium 4 プロセッサ担当アーキテクト。ワシントン大学にて 1985 年に電気工学の学士号、1990 年には同じく修士号を取得。IC 設計および CAD ツールの開発を長年手がけた後、ミシガン大学に入学、コンピュータ・アーキテクチャを研究。1994 年に博士号を取得後、インテルに入社。Pentium Pro および Pentium 4 プロセッサの開発に携わる。

電子メールアドレス：mike.upton@intel.com

インテルは、米国およびその他の国におけるインテル コーポ－レーションまたはその子会社の登録商標です。

Xeon および Intel486 は、米国およびその他の国におけるインテル コーポレーションまたはその子会社の商標です。

一般にブランド名または商品名は、各社の商標または登録商標です。

本稿は、http://www.intel.co.jp/jp/developer/index.htm より入手できます。

著作権、商標等について
http://www.intel.co.jp/sites/jp/tradmarx.htm

Copyright © Intel Corporation 2002.