スペキュレーティブ・プリコンピュテーション：マルチスレッディング・リソースの活用によるレイテンシの削減

Hong Wang -- インテル・ラボ、マイクロプロセッサ・リサーチ
Perry H. Wang -- インテル・ラボ、マイクロプロセッサ・リサーチ
Ross Dave Weldon -- インテル コーポレーション、ロジック・テクノロジ開発事業本部
Scott M. Ettinger -- インテル・ラボ、マイクロプロセッサ・リサーチ
Hideki Saito -- インテル コーポレーション、ソフトウェア・ソリューション事業本部
Milind Girkar -- インテル コーポレーション、ソフトウェア・ソリューション事業本部
Steve Shih-wei Liao -- インテル・ラボ、マイクロプロセッサ・リサーチ
John P. Shen -- インテル・ラボ、マイクロプロセッサ・リサーチ

検索用キーワード：キャッシュ・ミス、メモリ・プリフェッチ、プリコンピュテーション、マルチスレッディング、マイクロアーキテクチャ

摘要

スペキュレーティブ・プリコンピュテーション (SP) とは、本来シングルスレッド・アプリケーションでは有効に使われないマルチスレッディング・ハードウェア・リソースを利用してロングレンジのデータ・プリフェッチを積極的に行い、シングル・スレッド・アプリケーションのレイテンシを改善しようという手法です。SP では、シングル・スレッド・アプリケーションの明示的な並列化は行いません。SP の動作概要は次のとおりとなります。

- キャッシュ・ミスのペナルティによるパフォーマンス低下の原因のほとんどを占める、ごく一部のスタティック・ロード (**delinquent** ロード と呼ぶ) のみをターゲットとする。
- それぞれの delinquent ロードごとに、依存関係のある命令のみで構成されるスライスを特定する。
- ハードウェア・コンテキストに空きがあればスライスを動的に生成して、ロード・アドレスのプリコンピュテーションを投機的に行い、データをプリフェッチする。

こうすることで、キャッシュ・ミスの大部分は命令実行でオーバーラップできるため、元のプログラムにおけるクリティカル・パスからメモリ・レイテンシを隠蔽できるようになります。

基本的に、マルチスレッディング・マイクロアーキテクチャ技術が効果を発揮するのは、マルチタスク・ワークロードのスループットを高めたり、マルチスレッド化されたプログラムのパフォーマンスを改善する場合に限られるというのが従来の常識でしたが、SP はマルチスレッディング・ハードウェア・リソースを利用して一種の暗黙的なスレッド・レベルの並列化を行うことによって、シングル・スレッド・アプリケーションのパフォーマンスを大幅に向上できるという 1 つの可能性を示しています。従来の PC 環境におけるデスクトップ・アプリケーションのほとんどがシングルスレッド・コードで記述されており、マルチスレッディング・リソースを有効に活用できていないことを考えると、SP はこれらアプリケーションの並列化を手軽に行う手法として注目することができます。

本稿では SP に関するインテルの研究を振り返り、これまでに得られた主な成果を紹介します。まず最初は、インオーダーおよびアウトオブオーダーのマルチスレッド化されたマイクロアーキテクチャを使って、シミュレーション・ベースによる SP の評価を行っていた初期の研究から解説していきます。次に、製造前段階のハイパー・スレッディング・テクノロジ対応イン

テル® Xeon™ プロセッサ上でソフトウェア・ベースの SP (SSP) を適用し、ポインタを多用するアプリケーションのパフォーマンスを大幅に向上させることに成功した最近の実験についても紹介します。

はじめに

現在、プロセッサの高性能化において重大なボトルネックとなっているのが、メモリのレイテンシです。今日の大規模なアプリケーションでは、メモリ・アクセス・パターンの予測も難しくなっている上、ワーキング・セットのサイズもきわめて大きくなっているため、メモリへのアクセスが頻発します。もちろん、キャッシュ設計やプリフェッチ技術も進歩はしていますが、メモリがボトルネックになるという問題は解消されていません。特に、ポインタを多用するアプリケーションの場合は従来のストライド法によるプリフェッチが通用しないため、メモリ・レイテンシの問題はきわめて深刻になります。

これを解決する 1つの方法として、あるプログラムでメモリ・ストールが発生したら別プログラムの命令をオーバーラップさせて実行するという方法があります。こうすると、全体的なスループットで見るとシステムのパフォーマンスは効果的に改善されることになります。現在注目を集めている SMT (Simultaneous Multithreading) も、1つのプロセッサ上でいかに マルチタスク・ワークロードのスループットを改善するかを主な目的とした技術です [1][2][3]。SMT プロセッサでは、1サイクルで複数のハードウェア・コンテキスト (論理プロセッサ、またはハードウェア・スレッドとも呼ぶ) からスーパースケーラ・プロセッサの機能ユニットに対して命令を発行することができます。SMT はアーキテクチャの利用できる命令レベルの並列化を全体的に高めることで、総合的なスループットを改善します。このとき、各サイクルにおいて依存関係のないスレッドどうしに注目するという、ごく一般的な並列化を行います。

しかしこうした SMT の利用形態では、実行しているスレッドが 1つしかない場合はレイテンシの面で直接的なパフォーマンス向上が得られません。しかも、従来の PC 環境におけるデスクトップ・アプリケーションの大半はシングルスレッド・コードで記述されています。そこで、SMT の手法でレイテンシを削減することによってシングルスレッド・コードのパフォーマンスを向上させることは可能なのか、可能だとしたらどのような方法で実現するのか、といった点が重要な研究課題となっていました。

インテル・ラボでは、マルチスレッド化されたハードウェア・リソースを活用してシングルスレッド・アプリケーションのパフォーマンスを向上するための画期的な手法を、ハードウェア/ソフトウェアの両面から開発、評価すべく、マイクロアーキテクチャの研究を大規模に行ってきました。その 1つがスペキュレーティブ・プリコンピュテーション (SP) と呼ばれる、まったく新しいスレッド・ベースのキャッシュ・プリフェッチ・メカニズムです。SP の基本概念は、本来シングルスレッド・アプリケーションでは有効に使われないハードウェア・スレッド・コンテキストを利用してスペキュレーティブ (投機的) なスレッドを実行し、メインの (非スペキュレーティブな) スレッドを高速化しようというものです。スペキュレーティブ・スレッドは、実際にメイン・スレッドがキャッシュ・ミスを起こすよりもはるか前の段階であらかじめキャッシュ・ミス・イベントをトリガしておき、キャッシュ・ミス時のレイテンシを隠蔽することを目的としています。SP は、不規則で予測の難しいロード命令や、データ依存性の高いアクセス・パターンを示すロード命令などにターゲットを絞って効果を高めた特殊なプリフェッチ・メカニズムの一種とも考えられます。従来、これらのロードはハードウェアによるプリフェッチでも [5][6][7]、ソフトウェアによるプリフェッチでも [8] なかなか対処できませんでした。

今回の研究では、さまざまな面から SP の評価を行ってきました。当初は、インオーダーおよびアウトオブオーダーのマルチスレッド化された実験用プロセッサを用いて、シミュレーション・ベースによる SP の評価を行っていました [9][10][11][12][13][14] が、最近では、製造前段階のハイパー・スレッディング・テクノロジ対応インテル® Xeon™ プロセッサ上でソフトウェア・ベースの SP (SSP) を実験し、ポインタを多用するベンチマークのパフォーマンスを大幅に向上させることに成功しました。本稿では、こうした流れにそって研究成果を紹介していきます。

まず最初に、SP の背景にある原理を説明し、次に SP の基本的なアルゴリズムや、SP の効果を高める連鎖トリガなどの最適化手法について解説します。さらに、従来のレイテンシ隠蔽手法であるアウトオブオーダー実行と SP を比較し、これら 2つの手法を組み合わせた場合の効果についても考察します。また、ハードウェア・ベースの SP およびソフトウェア・ベースの SP (SSP) について、それぞれのトレードオフを検証し、特に、バイナリを自動的に SSP 化する post-pass コンパイル・ツールについても紹介します。このツールを利用すると、手動で最適化した SSP にも匹敵するパフォーマンス向上効果が得られます。さらに、製造前段階のハイパー・スレッディング・テクノロジ対応インテル Xeon プロセッサを使用し、SSP を適用することによってアプリケーションの高速化に成功した最近

の実験についても紹介します。最後に、その他の関連研究についても取り上げます。

スペキュレーティブ・プリコンピュテーションの基本概念

もともと、スペキュレーティブ・プリコンピュテーション (SP) の基本概念は、ハイパー・スレッディング・テクノロジ対応インテル® Xeon™ プロセッサのシリコンが入手可能になる以前から研究が始められていました。当初は、SMT (Simultaneous Multithreading) をサポートし、インオーダーまたはアウトオブオーダーのいずれにも設定できるパイプラインを備えた実験用の Itanium™ プロセッサをモデル化したシミュレーション・インフラストラクチャを使って研究を行っていました。ハードウェア・ベース、ソフトウェア・ベースで SP を実装した際のそれぞれのトレードオフについては後で論じるとして、ここでは表 1 に示した実験用プロセッサ・モデルをまず使用してみます。また、今回の研究ではベンチマークとして SPEC2000 および Olden スイートから art、equake、gzip、mcf、health、mst を選んで使用しました。

表 1：実験用 Itanium プロセッサ・モデルの詳細

パイプライン構造	インオーダー：8〜12 ステージのパイプライン アウトオブオーダー：12〜16 ステージのパイプライン
フェッチ	1 スレッドから 2 バンドル、または 2 スレッドから 1 バンドルずつ
分岐予測機構	2K エントリ GSHARE。256 エントリ、4 ウェイ
拡張	スレッドごとにプライベートなインオーダー8 バンドル拡張キュー
レジスタ・ファイル	スレッドごとにプライベートなレジスタ・ファイル。 128 の整数レジスタ、128 の浮動小数点レジスタ、64 のプレディケート・レジスタ、128 のアプリケーション・レジスタ
実行時の帯域幅	インオーダー：1 スレッドから 6 命令、または 2 スレッドからそれぞれ 3 命令ずつ アウトオブオーダー：18 命令のスケジュール・ウィンドウ
キャッシュ構造	L1 (命令用とデータ用にそれぞれ)：16K 4 ウェイ、8 ウェイ・バンク、1〜2 サイクル L2 (共有): 256K 4 ウェイ、8 ウェイ・バンク、7〜14 サイクル L3 (共有): 3072K 12 ウェイ、1 ウェイ・バンク、15〜30 サイクル フィル・バッファ (MSHR)：16 エントリ。全キャッシュとも 64 バイト・ライン
メモリ	115〜230 サイクルのレイテンシ、TLB ミス時のペナルティ = 30 サイクル

Delinquent ロード

ほとんどのプログラムでは、ごく一部のスタティック・ロードがキャッシュ・ミスの大半を占めています [15]。図 1 は、表 1 に示したプロセッサ・モデル上でベンチマークを実行し、L1 データ・キャッシュ・ミスの多い上位 50 個のスタティック・ロードの累計をグラフにしたものです。これを見ても明らかなように、これらのプログラムではごく少数のスタティック・ロードがキャッシュ・ミスの大部分を占めています。このようなロードをここでは delinquent ロードと呼ぶことにします。

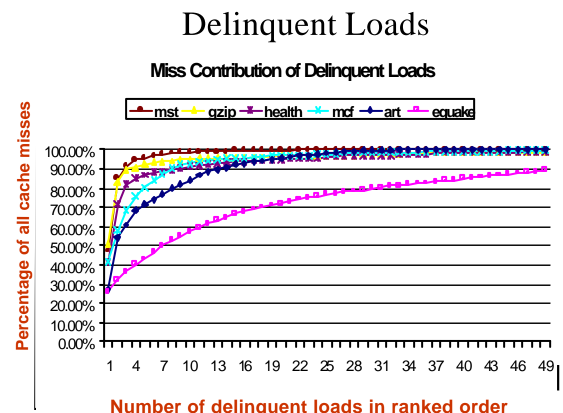

図 1：delinquent ロードによる L1 データ・キャッシュ・ミスの累計

これらのロードがパフォーマンスに与える影響を調べるため、図 2 ではすべてのロードが L1 キャッシュにヒットする完全なメモリ・サブシステムと、キャッシュ・ミスの原因となっている上位 10 個の delinquent ロードが常に L1 キャッシュにヒットするようにしたメモリ・サブシステムで、それぞれパフォーマンスがどれだけ向上したかを比較しています。これを見ると、ほとんどの場合、最も影響の大きい delinquent ロード

によるパフォーマンス・ロスを排除するだけで、完全なメモリにかなり近い性能向上が実現しています。つまり、ごく一部の delinquent ロードのレイテンシを削減する方法を見つければ、それだけで大幅なパフォーマンス向上が実現することになります。

Performance Impact of D-Loads

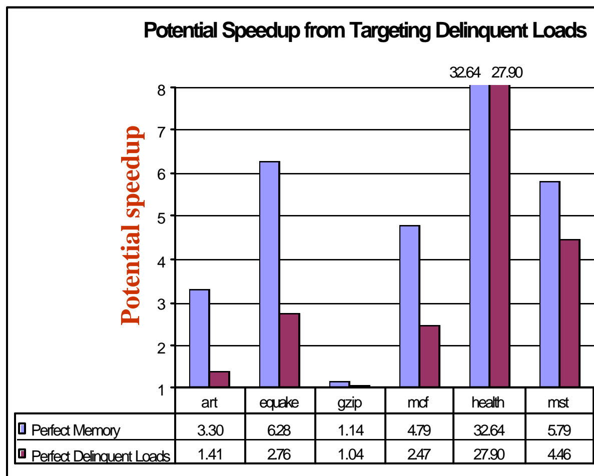

図 2：10 個の delinquent ロードが常に L1 キャッシュにヒットするようにした場合の性能向上

SP の概要

delinquent ロードのプリフェッチを効果的に行うため、スペキュレーティブ・プリコンピュテーション (SP) ではプリコンピュテーション・スライス (p-スライス) を作成します。p-スライスとは、delinquent ロードがどのアドレスにアクセスするかを計算する一連の (依存関係にある) 命令で構成されます。トリガとなるイベントによって p-スライスが呼び出されると、スペキュレーティブ・スレッドが生成されて p-スライスを実行します。こうして投機的に実行された p-スライスは次に、メインのスレッドが後で実行するはずの delinquent ロードをプリフェッチしておきます。SP では、基本トリガ (メイン・スレッド内の指定した命令がリタイアした時に発生)、または連鎖トリガ (スペキュレーティブ・スレッドが明示的に別のスペキュレーティブ・スレッドを生成した場合に発生) のいずれかを検出した場合にスペキュレーティブ・スレッドが生成されます。

スペキュレーティブ・スレッドを生成する際には、ハードウェア・スレッド・コンテキストをスレッドに割り当てる、必要な live-in バリューをレジスタ・ファイルにコピーする、スレッド・コンテキストに対して p-スライスの最初の命令のアドレスを提供する、という一連の動作が行われます。すべてのハードウェア・コンテキストが使用中の場合は、スペキュレーティブ・スレッド生成のリクエストは無視されます。

スペキュレーティブ・スレッドが生成されると、必要な live-in バリューは必ずスレッド・コンテキストにコピーされます。これによって、子スレッドがレジスタの値を読み出す前に別のスレッドが値を上書きしてしまうというスレッド間のハザードを防ぎます。幸い、表 2 に示したとおり、コピーが必要となる live-in バリューの数はそれほど多くありません。

表 2：スライスに関する統計

ベンチマーク	スライス数	平均サイズ (命令数)	live-in の平均数
art	2	4	3.5
equake	8	12.5	4.5
gzip	9	9.5	6.0
mcf	6	5.8	2.5
health	8	9.1	5.3
mst	8	26	4.7

スペキュレーティブ・スレッドは、生成されてから p-スライス中のすべての命令の実行が完了するまでの間、ハードウェア・スレッド・コンテキストを 1 つ占有します。また、スペキュレーティブ・スレッドはアーキテクチャ・ステートを更新することはできません。特に、p-スライス中のストアに関しては、メモリの状態を一切更新してはならないことになっています。ただし、今回の実験で行ったベンチマークに関しては、ストア命令を含む p-スライスはありませんでした。

SP におけるタスク

SP を利用するには、delinquent ロードを特定する、これらのロードに対して p-スライスを作成する、トリガを埋め込む、といったいくつかのタスクを行う必要があります。さらに、SP を使って動的な実行を行う際には、プリコンピュテーションによるプリフェッチを「適時性」と「正確性」の面で適切に制御する必要もあります。以上述べたタスクは、さまざまなアプローチで行えます。例えば、補助的にコンパイラを使用する場合、ハードウェアを使用する場合、およびソフトウェア/ハードウェアの両アプローチを併用する場合などです。また、これらの手順は、SMT をサポートしたプロセッサであれば、命令セット・アーキテクチャ (ISA) やパイプライン構成の違いには関係なく、どのプロセッサにも応用できます。SP のさまざまな実装方法については、別のセクションでさらに詳しく解説します。

Delinquent ロードの特定

キャッシュ・ミスの原因の大半を占める delinquent ロードを特定するには、メモリ・アクセスのプロファイリングを行います。これは、コンパイラやメモリ・アクセス・シミュレータ [15] を用いる場合や、VTune™ パフォーマンス・アナライザ [16] など、実際のシリコンを対象にした専用のプロファイリング・ツールを用いる場合があります。このようなプロファイル分析を行ってみて、パフォーマンスに与える影響 (レイテンシ) の大きいロードがあれば、それが delinquent ロードです。なお、delinquent ロードを特定する際の評価基準として L1 キャッシュ・ミスの合計回数を使うこともあれば、L2 または L3 キャッシュ・ミス、あるいは全体的なメモリ・レイテンシなど他の基準で判断することもあります。今回の研究でも、シミュレーション・ベースの研究では delinquent ロードの特定を行う際の基準として L1 キャッシュ・ミスを使用していますが、製造前段階のハイパー・スレッディング・テクノロジ対応インテル Xeon プロセッサを用いた実験では、VTune パフォーマンス・アナライザによる L2 キャッシュ・ミスのプロファイリングをもとに delinquent ロードの特定を行っています。

P-スライスの作成と最適化

次に、各 delinquent ロードごとに p-スライスを作成します。p-スライスの作成手段は環境によってさまざまで、手作業で行う場合、シミュレータを使う場合 [11] [13]、コンパイラを使う場合 [14]、あるいはハードウェアで直接行う場合 [12] などがあります。例えば、基本トリガを含む p-スライスの場合、動的な命令トレースのウィンドウ内で後方スライシング [17] を行うという従来の手法で作成することができます。delinquent ロードが依存していない命令を除外すれば、一般に p-スライスのサイズは 1 スライスあたり 5～15 命令程度のきわめて小さいサイズに抑えることができます。連鎖トリガを含む p-スライスでは、さらに複雑なプロセスを使って p-スライスの作成を行う必要があります。

一般に、連鎖トリガを含む p-スライスは、(1) プロローグ、(2) 子スレッド生成命令 (さらに別の p-スライスを生成する)、(3) エピローグの 3 つの部分で構成されます。プロローグは、ループ内の依存関係 (ループ誘導変数に対して更新を行うなど、あるループ・イタレーションで生成された値を次のループ・イタレーションで使うような場合) に関連した値を計算する命令で構成されます。エピローグは、ターゲットとなる delinquent ロードに対するアドレスを生成する命令で構成されます。連鎖トリガを作成する目的は、プロローグをできるだけ高速に実行することにあります。そうすることで、別のスペキュレーティブ・スレッドを次々と生成できるようになります。

ループ内の delinquent ロードをターゲットとした p-スライスに連鎖トリガを追加するには、基本トリガを使った p-スライスを抽出するアルゴリズムをさらに強化して、1 つの delinquent ロードの異なるインスタンス間の距離を追跡するようにします。同一の p-スライスの 2 つのインスタンスが、固定されたサイズの命令ウィンドウ内に一貫して生成される場合は、同じ delinquent ロードをターゲットとする連鎖トリガを含む p-スライスを新たに作成します。あるスライスの命令が次の p-スライスで使われる値を変更するような場合には、この命令はプロローグに追加されます。delinquent ロードによってロードされたアドレスを生成する必要のある命令はエピローグに追加されます。プロローグとエピローグの間には子スレッド生成用の命令が挿入され、この p-スライスと同じものが子スレッドとして生成されます。

プリコンピュテーションの調整

SP ベースのプリフェッチが効果を発揮するためには、「正確性」と「適時性」が要求されます。「正確性」とは、p-スライスが生成されたらなるべく有効な live-in バリューを使うようにして、正確なプリフェッチ・アドレスを生成できるようにすることです。また、「適時性」とは、プリフェッチを行っているスペキュレーティブ・スレッドがメイン・スレッドの実行よりも遅れたり、あるいはあまりにも早く実行されることがないようにすることを意味します。

正確性については、トリガがプロセッサ・パイプラインのコミット・ステージに到達してからスペキュレーティブ・スレッドの生成が行われるのであれば、関連する p-スライスの live-in バリューは通常アーキテクチャ的に正しいことが保証されるため、プリコンピュテーションは必ず正しいプリフェッチ・アドレスを生成することになります。一方、トリガ命令がパイプラインのデコード・ステージで検出された段階でスペキュレーティブ・スレッドの生成を試みるというアプローチもあります。しかし、このように早い段階でスペキュレーティブ・スレッドの生成を行うと、トリガおよび live-in バリューの両方がまだ投機的であり、不正なアドレスからプリフェッチを行ってしまう可能性があるという欠点があります。

適時性については、基本トリガの定義が、トリガと目的とする delinquent ロード間の距離に大きく関係していることが挙げられます。これは、子スレッドの生成はメイン・スレッドの進行に密接につながっているためです。子スレッド生成に伴うオーバーヘッドが発生すると、プリフェッチの効果が損なわれるばかりでな

く、メイン・スレッドに対して余分なレイテンシを引き起こしてしまうことにもなります。

一方、連鎖トリガの場合はスレッドの生成をメイン・スレッドの進行から切り離して行えるという利点はありますが、過剰にプリフェッチを行ってしまい、キャッシュ内の有用なデータをメイン・スレッドがアクセスする前に追い出してしまう可能性があります。そこで、メイン・スレッドと SP スレッドが適切な距離を保って実行できるようにするため、**OSC (Outstanding Slice Counter)** と呼ばれる機構を用意して、生成されたスペキュレーティブ・スレッドの数と、メイン・スレッドによってまだリタイアされていない delinquent ロードのインスタンスの数の相対的なバランスを、各 delinquent ロードのサブセットごとに追跡するようにしています。OSC の追跡機構の各エントリにはカウンタ、delinquent ロードの命令ポインタ (IP)、および p-スライスの最初の命令のアドレス (これによって p-スライスを識別) が格納されます。カウンタの値はメイン・スレッドが delinquent ロードをリタイアすると減り、p-スライスが生成されると増える仕組みになっています。スペキュレーティブ・スレッドが生成されても、OSC 内のエントリのカウンタが負の数の場合は、スペキュレーティブ・スレッドはカウンタの値が正になるまで保留状態のまま待たなければなりません。この間はハードウェア・スレッド・コンテキストへの割り当てが行われません。

後述のとおり、こうした制御メカニズムは、スペキュレーティブ・スレッドの一部として完全にソフトウェアのみで実装することもできます。

SP のトレードオフ

今回 SP に関する研究を行った結果、1つの事実が判明しました。それは、理想的なハードウェア環境を想定した場合の基本トリガと、ハードウェア・サポートに制約が多いながらも適正な制御メカニズムを採用した連鎖トリガを比べると、後者の方がはるかに効果が大きいという点です。以下、基本トリガと連鎖トリガのトレードオフをまとめてみました。

理想的なハードウェア環境での基本トリガ

図 3 は、2 種類のほぼ理想的な SP 設定によってパフォーマンスがどれだけ向上したかを示したものです。1 つはリネーム・ステージの段階でメイン・スレッドからスペキュレーティブ・スレッドを生成するという積極的な設定にしてあります。ただし、必ず正しいコントロール・フロー・パスにある命令がスレッドを生成するものと仮定しています。もう 1 つは、命令がコミット・ステージに達して、正しいパスにあることが保証された時点でスペキュレーティブ・スレッドを生成するという、消極的な設定にしています。いずれの場合も、ハードウェア環境はきわめて理想的な状態を想定しており、メインの親スレッドのコンテキストから子スレッドのコンテキストに live-in バリューを直接コピー (1 サイクルでのフラッシュ・コピー) できるようになっています。このため、スペキュレーティブ・スレッドが生成されてから 1 サイクル後には p-スライスの実行を開始できることになります。

図 3 では、ハードウェア・スレッド・コンテキストの総数を 2、4、8 と変えてベンチマークを行い、リネーム・ステージでスペキュレーティブ・スレッドを生成した場合 (左側) と、コミット・ステージで生成した場合 (右側) の高速化のようすを比較しています。

現実的なハードウェア環境での基本トリガ

次に、より現実的な環境で SP を行ってみることにします。ここでは、スレッドの生成をトリガ命令がリタイアしてから行うことにするほか、パイプライン・フラッシュの可能性や、メモリを介して live-in バリューを転送する際に 2 サイクル以上を要するなどのオーバーヘッドについても想定しています。このアプローチは、理想的なハードウェア・アプローチとは 2 つの点で異なります。1 つは、スレッドの生成がもはや瞬時には行われないという点です。ハンドラ・コードを呼び出して実行し、ハードウェア・スレッドが利用可能かどうかをチェックし、live-in バリューをメモリに書き出してスレッド間転送を行うなどの必要があるため、メイン・スレッドの速度が低下してしまいます。少なくとも、このハンドラを呼び出すには一度パイプラインをフラッシュしなければなりません。もう 1 つの相違点は、転送メモリ・バッファから live-in バリューを最初にロードするためのプロローグを p-スライスに追加しなければならないため、プリコンピュテーションの開始が遅れてしまうという点です。

Potential Speed-up (Basic Triggers)

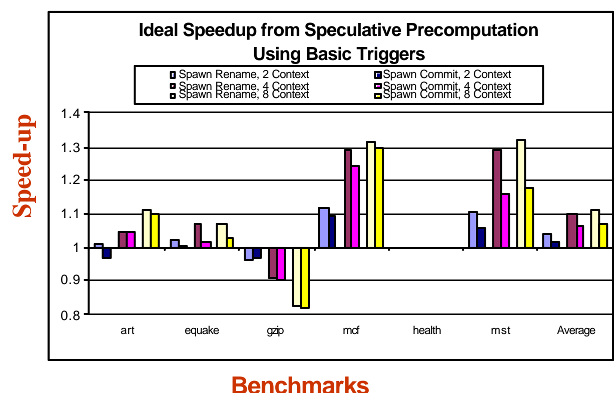

図 3：理想的なハードウェア環境で基本トリガを使用した場合の SP による性能向上

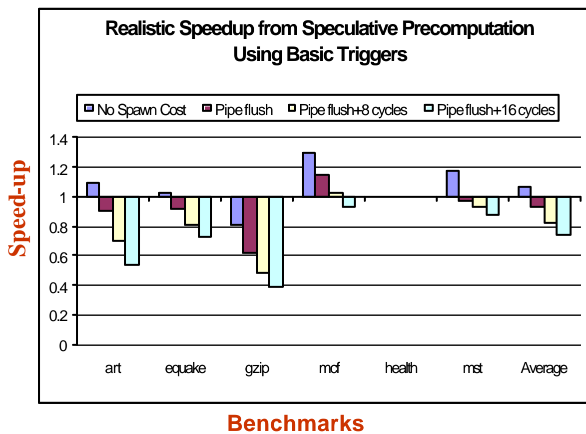

図 4：現実的なハードウェア環境で基本トリガを使用した場合の SP による性能向上

図 4は、より現実的なハードウェア環境において、8 つのハードウェア・スレッド・コンテキストを備えたプロセッサを使い、どれだけの性能向上が得られるかを示したものです。プロセッサについてはスレッド生成コストのそれぞれ異なる 4種類の設定を行っています。一番左の設定はリファレンス用で、スペキュレーティブ・スレッドの生成時にメイン・スレッドに対するペナルティがまったく発生しないようにしています。ただし live-in バリューを転送メモリ・バッファから読み出すために一連のロード命令は実行しなければなりません。この設定では、メイン・スレッドがスペキュレーティブ・スレッドを瞬時に生成できるため、最も高いパフォーマンスが得られます。他の 3つの設定では、スペキュレーティブ・スレッドを生成すると、メイン・スレッドにおいてトリガ以降の命令がパイプラインからフラッシュされてしまいます。左から 2番目の設定では、ペナルティとなるのはこのパイプラインのフラッシュのみですが、3番目と 4番目の設定では、live-in バリューの転送を行うためのハンドラ・コードを実行しなければならないため、それぞれ 8サイクル、16サイクルのペナルティが余分に生じるようになっています。

この結果を、理想的なハードウェアを使用した場合の SP のパフォーマンス (図 3) と比べてみると、現実的なハードウェア環境での SP はかなり不本意な結果に終わっています。その主な原因は、メイン・スレッドがスペキュレーティブ・スレッドを生成する際に発生するオーバーヘッドにあります。特に、パイプライン・フラッシュによるペナルティと、ハンドラ内で live-in バリューを転送する命令の実行コストの 2つが、メイン・スレッドのパフォーマンスに悪い影響を与えています。

連鎖トリガ

図 5は、現実的なハードウェア環境で、連鎖トリガを使った SP の性能向上を、スレッド・コンテキストの数を変えて実験した結果を示したものです。ここでは、スレッド生成時にはパイプライン・フラッシュが起こり、さらに 16サイクルの追加ペナルティが発生すると仮定しています。連鎖トリガはメモリの並列化が十分に存在する場合はスレッド・コンテキストを有効に利用できるため、平均すると 4スレッド時で 51%、8 スレッド時で 76% という圧倒的な高速化を達成しています。

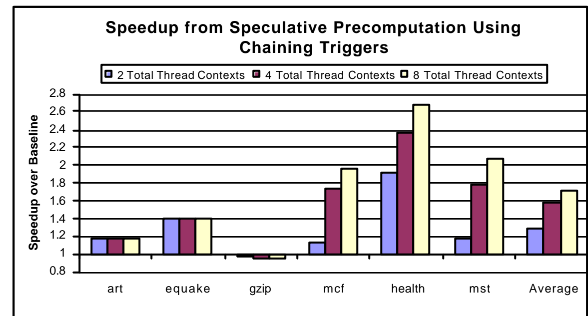

図 5：現実的なハードウェア環境で連鎖トリガを使用した場合の SP による性能向上

特にパフォーマンスの伸びが大きいのが、health です。基本トリガの場合はほとんどパフォーマンスが向上していない (図 4) のに対し、連鎖トリガの場合は 169% もの高速化を達成しています。

図 6は、SP なしのベースライン・プロセッサ、8つのスレッド・コンテキストを備えたプロセッサで基本トリガを使用した場合、8つのスレッド・コンテキストを備えたプロセッサで基本トリガと連鎖トリガの両方を使用した場合の 3つのプロセッサ設定を行い、delinquent ロードによるメモリ・アクセス状況を階層別に示したものです。

Sources of Speed-up

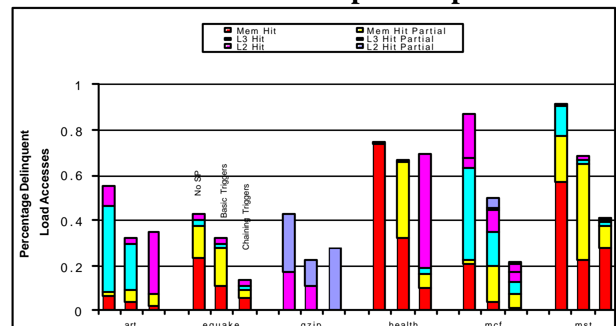

図 6：SP ベースのプリフェッチによる、各メモリ階層におけるキャッシュ・ミスの削減

一般に、基本トリガは高い正確性を実現できますが、メイン・メモリへのアクセスを必要とするロードについてはそれほど削減率が大きくありません。基本トリガは、L1 キャッシュ・ミスなど比較的レイテンシの低い delinquent ロードをターゲットにした場合には効果を発揮しますが、メイン・メモリへのアクセスが必要となるようなキャッシュ・ミスの場合は、適時性の面で満足のいくプリフェッチが行えません。

一方、連鎖トリガは広範にわたってレイテンシの削減に成功しており、メイン・メモリへのアクセスを必要とするデータについても、基本トリガに比べはるかに適時性に優れたデータ・プリフェッチを行えます。これは、連鎖トリガの場合は delinquent ロードの特定が効果的に行え、メイン・スレッドよりもはるかに前の段階でプリフェッチが行えるためです。

メモリ・レイテンシの隠蔽に関する SP と OOO の比較

スペキュレーティブ・プリコンピュテーション (SP) のようなスレッド・ベースのプリフェッチ手法が登場する以前は、キャッシュ・ミスのレイテンシを隠蔽するマイクロアーキテクチャ手法としては主にアウトオブオーダー (OOO) 実行が使われてきました [18][19][20]。OOO プロセッサにはレジスタ・リネーム機構とスケジューラが用意されており、パイプライン内で流れている命令を動的にスケジューリングし、キャッシュ・ミスによるロード待ちの間にそのロードとは依存関係のない命令を次々と実行していきます。

基本的に、OOO と SP はいずれもキャッシュ・ミスによるレイテンシと命令実行をオーバーラップさせることによってメモリ・レイテンシを隠蔽することを目的としています。OOO の場合は、ロード時にキャッシュ・ミスが起こると、そのロードとは依存関係のない命令を探し、キャッシュ・ミスによるサイクルの間にこれらの命令をオーバーラップ実行します。一方、SP ではメイン・スレッドよりもきわめて早い段階で delinquent ロードのプリフェッチを行います。これは、メイン・スレッドの現在の命令実行と将来のキャッシュ・ミスをオーバーラップさせたものといえます。

SP と OOO はいずれもプログラムのクリティカル・パスで発生するデータ・キャッシュ・ミスのペナルティを軽減できますが、ターゲットとするメモリ・アクセス命令や、どのキャッシュ階層に効果的かといった点で異なります。OOO の場合、可能性としてはキャッシュ構造の全階層に対するロード/ストア命令のキャッシュ・ミス・ペナルティを隠蔽できますが、中でも最も効果が高いのは L1 キャッシュ・ミスによるペナルティの隠蔽です。L2 または L3 キャッシュ・ミスに関してはレイテンシがきわめて大きいため、その間に実行できる十分な命令を見つけることが難しい場合があります。一方、SP ではメモリへのアクセスを必要とするような、レイテンシの大きいキャッシュ・ミスを引き起こすごく一部の delinquent ロードにターゲットを絞っています。

SP と OOO の違いを数値化するため、ここでは表 1 で示した実験用プロセッサ・モデルを用いて 2 つのタイプのベンチマークの評価を行いました。1 つは、CPU 負荷の高いワークロード (SPEC2000Int の gap、gzip、parser) で、もう 1 つはメモリ・アクセスの多いワークロード (SPEC2000fp の equake、SPEC2000int の mcf、Olden スイートの health) です。

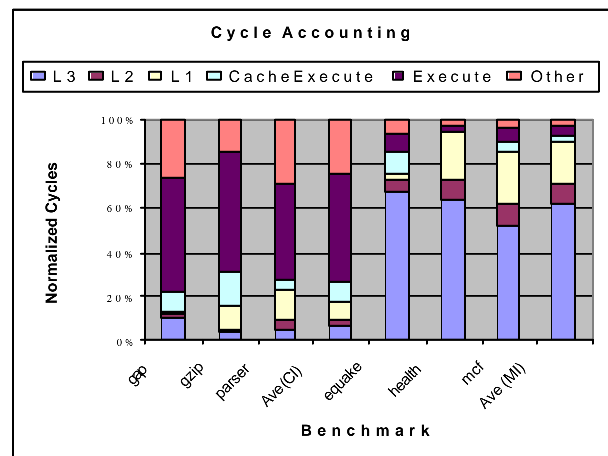

図 7：CPU 負荷の高いワークロードと
メモリ・アクセスの多いワークロードを
インオーダー・マシン上で実行した結果

図 7 は、インオーダーのベースライン・プロセッサで上記のベンチマークを実行した場合のサイクルの内訳を示したものです。L1、L2、L3 と表記されているのは、それぞれの階層でキャッシュ・ミスが発生した際にメモリ・サブシステムにアクセスしているサイクルを表します。Execute は、メモリ・サブシステムのアイドル時にプロセッサが命令を発行して実行しているサイクルを表します。CacheExecute はキャッシュ・ミスと命令実行をオーバーラップさせているサイクルです。図 7 を見ると、CPU 負荷の高いベンチマークでは明らかに Execute の比率が大きくなっている一方、メモリ・アクセスの多いベンチマークではキャッシュ・ミス時の待ち時間が占める割合が大きくなっています。

図 8 は、SP、OOO、およびその両方の組み合わせによって、ベースライン・モデルに比べどれだけの高速化が実現するかを示したものです。OOO プロセッサ・モデルには 4 つのパイプ・ステージが追加されており、より複雑な処理が行えるようになっています。

SP に関しては、連鎖トリガを使用し、プリコンピュテーションの調整も行っているものと仮定しています。

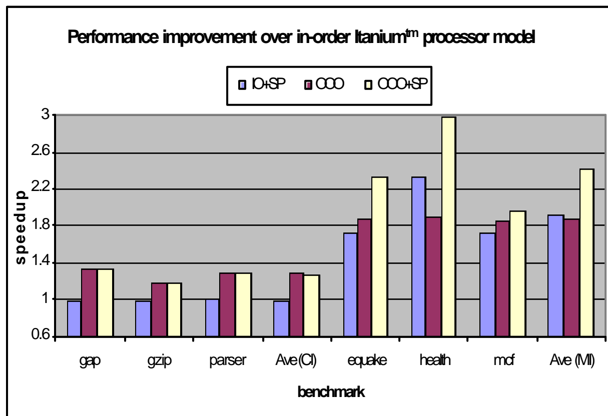

図 8：インオーダーと SP、OOO のみ、OOO と SP による高速化 (インオーダーのみの場合と比較)

図 9 は、インオーダー実行のみの場合を 100% としたサイクルの内訳を示しています。これを見ると、レイテンシを削減することによってどの部分のサイクルが高速化しているかが分かります。

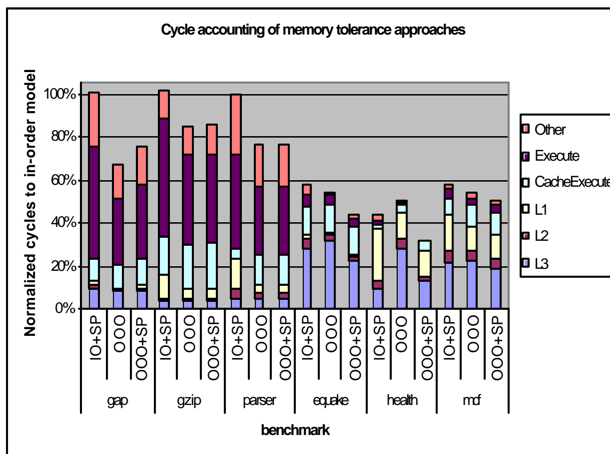

図 9：インオーダーと SP、OOO のみ、OOO と SP のサイクルの内訳 (インオーダーのみの場合を 100% とする)

以上の実験結果をふまえ、その内容を以下に要約します。

OOO と SP の比較

SP は L2 または L3 キャッシュ・ミスを頻繁に起こす delinquent ロードの上位 10 個のみをターゲットとしたものですが、図 8 でメモリ・アクセスの多いワークロードの平均結果を見ると、インオーダーと SP の組み合わせの方が OOO よりも速度向上がわずかに大きくなっています。これは、図 9 でも分かるとおり、SP と OOO ではキャッシュ階層の異なる部分でキャッシュ・ミス時のペナルティが削減されているためです。例えば health の場合、ベースラインのインオーダー・マシンでは L3 のサイクル数が 62% を占めていましたが、OOO では 28%、SP では 9% にまで低下しています。

しかし CPU 負荷の高いベンチマークの場合は、SP ではむしろパフォーマンスが低下することがあります。というのは、これらのベンチマークでは delinquent ロードで L1 キャッシュ・ミスが発生してもそのほとんどは L2 キャッシュにヒットするため、SP スレッドを投機実行してタイムリーにプリフェッチを行うことによる効果が現れにくいためです。しかも、子スレッドを生成することによって、メイン・スレッドとの間でリソースの競合が発生しやすくなるため、メイン・スレッドのパフォーマンスが低下する可能性も出てきます。

一方、OOO はどの階層でキャッシュ・ミスが発生しても、一定の効果が上がっています。また、機能ユニットにおけるレイテンシの大きい実行にも効果があります。例えば parser の場合、OOO では L1 キャッシュのストール・サイクルを 10% 削減しているほか、Execute で表される実行サイクルも 12% 削減しています。さらに、キャッシュ・ミスと命令実行をオーバーラップさせている CacheExecute の部分も、9% 増加しています。

OOO と SP の組み合わせ

図 8 を見ると、CPU 負荷の高いベンチマークでは、OOO に SP を組み合わせても OOO 単独の場合に比べてほとんど高速化が認められません。

しかしメモリ負荷の高いベンチマークの場合では、ベンチマークの種類によって OOO と SP の組み合わせによる効果もさまざまです。例えば health の場合、OOO と SP をそれぞれ単独で使用するとそれぞれ約 131% と 90% の速度向上が実現しています。ところが SP と OOO を組み合わせると 198% もの性能向上を達成しています。これは、SP と OOO の潜在的な相補効果の大きさを示しています。図 9 を見ると、この効果がどこから来たものかが分かります。health では、SP のみの場合 L3 サイクルは 9% にまで削減されていますが、L1 サイクルは改善されていません。一方、OOO のみの場合は L1 サイクルが 11% にまで削減されていますが、L3 サイクルについては削減率はあまり大きくありません。つまり、SP と OOO を組み合わせると L1 と L3 キャッシュの両方で効果が期待できるため、L1 と L3 の割合が一挙に削減されることになります。このように、それぞれが明らかに異なる階層でのキャッシュ・ミスに対処するというのが、OOO と SP の相補効果の原理です。また、SP と OOO を組み合わ

せた場合、ほとんどすべての命令実行がメモリ・アクセスとオーバーラップできており、メモリ・レイテンシ隠蔽の手法としてはきわめて望ましい効果が得られています。

一方、図 9 に示したように、mcf の場合は SP を用いたインオーダー実行 (IO+SP) と OOO のみの場合を比べると、サイクルの内訳にあまり大きな差が認められません。これは、ループ・ボディ内の delinquent ロードを SP と OOO が二重にターゲットにしているため、両者の効果が重複していることを示します。

このように、OOO プロセッサ上で効果的に SP を利用するには、これら 2 つのアプローチの効果が重複しないように注意することが重要になります。特に、典型的なメモリ負荷の高いループでは、ポインタを含む冗長なループ・コントロールは通常 OOO プロセッサのクリティカル・パス上にあります。ループ・コントロールは、ループ内の依存関係を解消する命令や次のループ・イタレーションに対する誘導変数を計算する命令で構成されます。ループ・コントロール内でこれらの計算が完了すると、複数のイタレーションから依存関係のない命令を効果的に実行することで、特定のイタレーションのループ・ボディで発生したキャッシュ・ミスを隠蔽することができます。SP と OOO を上手に組み合わせるには、SP はループ・コントロール内でレイテンシの大きいロードのプリフェッチを行わせ、OOO はループ・ボディ内の delinquent ロードのレイテンシを隠蔽させるようにするのが賢明な方法といえます。こうすることによって、health のような理想的な相補効果が得られることになります。

ハードウェア単独の SP とソフトウェア単独の SP の比較

スペキュレーティブ・プリコンピュテーション (SP) の基本的な手順やアルゴリズムは、ハードウェアのみによるアプローチ [12]、ソフトウェアのみによるアプローチ [14]、さらにはこれらの混合型アプローチ [11][13] など、さまざまな手法で実装できます。

まず一方の極にあるのが、ハードウェアのみによるアプローチです。インテルでは、カリフォルニア大学サンディエゴ校の Tullsen 教授の研究チームとの緊密なコラボレーションを通じて、ハードウェア単独の SP について研究しました。これは「ダイナミック・スペキュレーティブ・プリコンピュテーション (DSP)」と呼ぶもので、ハードウェア・メカニズムを用いてプログラムの delinquent ロードを実行時に特定し、これらロードのプリフェッチを行うためのプリコンピュテーション・スライスを生成するという手法です。スレッド・ベースのプリフェッチ同様、プリフェッチ・コードはメイン・プログラムから切り離されるため、従来のソフトウェアによるプリフェッチよりもはるかに高い柔軟性が得られます。ハードウェア・プリフェッチ同様、DSP はレガシー・コード上でも動作し、将来のアーキテクチャとのソフトウェア互換性も維持できます。また、動的な情報をもとにしてプリフェッチを開始したり、プリフェッチの効果を評価できるという利点もあります。しかしソフトウェア・アプローチとは異なり、DSP でのスペキュレーティブ・スレッドでは、スライスの作成、スレッドの生成、拡張、および削除などはすべてハードウェアで行います。基本トリガおよび連鎖トリガ・ベースの p-スライスは、クリティカル・パスから分離したバックエンド機構を使うことで効率よく作成できます。p-スライスの最適化はほとんど行わなくても、メモリ負荷の高い各種ベンチマークにおいて 14% もの性能向上が得られます。さらに積極的に p-スライスの最適化を行えば、平均で 33% も性能が向上します。

興味深いのは、いくつもの非スペキュレーティブなスレッドが同時に実行される通常のマルチスレッド環境においても、頻繁にキャッシュ・ミスのペナルティを引き起こすロードに対して SP の手法を適用すると、SP から直接的に恩恵を受けるのがそのうちの 1 つのスレッドであっても全体的なスループットが実際に向上しているという点です。つまり、SP は本来シングルスレッド・アプリケーションのレイテンシを低減することを目的としているにもかかわらず、マルチプログラミング環境においてもスループット向上に貢献できるということです。

もう一方の極にあるのがソフトウェアのみによるアプローチです。われわれは既存のシングルスレッド・バイナリをマルチスレッド・プロセッサ上で実行できるよう自動的に SSP 化するという post-pass コンパイル・ツールを開発しました [14]。これには専用のハードウェア・メカニズムは一切必要ありません。このツールはインテルの IPF プロダクション・コンパイラ・インフラストラクチャに実装されており、次のタスクを実行できます。

1) 既存のシングルスレッド・バイナリを解析してプリフェッチ・スレッドを生成する。

2) トリガ・ポイントを特定してオリジナルのバイナリ・コードに埋め込む。

3) プリフェッチ・スレッドを付け加えた新しいバイナリを作成する。プリフェッチ・スレッドは実行時に子スレッドとして生成される。

新しく作成されたバイナリを実行すると、プリフェッチ・スレッドが生成され、メイン・スレッドと並行して実行されます。初期の実験結果でも、スペキュレー

ティブ・スレッドでプリフェッチを行うと、インオーダー・プロセッサ上でポインタを多用するベンチマークを行った場合、16% ～104% という圧倒的な高速化を実現しています。さらに、上記のバイナリ自動 SSP 化ツールで作成したコードと、手作業で生成した SSP コードを同じプロセッサ上で実行すると、後者の方が高速化の度合いは高いものの、その差は最大でも 18% しか認められませんでした。

われわれの知る範囲において、このようにバイナリを自動的に SSP 化するツールが実装され、しかもこのツールを使って、ターゲットとなる delinquent ロードに関連した命令スライスの抽出、適切な子スレッド生成ポイントの特定、効果的なプリフェッチに必要なスレッド間通信の管理によるタイムリーなプリエグゼキューションなど、すべてのプロセスを効果的に行うことに成功したのはこれが初めての事例です。

ハイパー・スレッディング・テクノロジ対応インテル® Xeon™ プロセッサにおける SP

ハイパー・スレッディング・テクノロジ対応インテル® Xeon™ プロセッサのシリコンが入手できたことにより、これまで主にシミュレーション・ベースの実験用プロセッサ・モデル上で開発を行ってきたスペキュレーティブ・プリコンピュテーション (SP) のアイディアを、実際のコンピュータで試してみることが可能になりました。製造前段階のハイパー・スレッディング・テクノロジ対応インテル Xeon プロセッサを搭載したシステムを入手してから数週間後には、重要なアイディアや画期的手法をもとに、ポインタを多用するベンチマークのコードに手作業でソフトウェアのみによる SP (SSP) を施し、大幅な高速化を実現することができました (表 3)。ベンチマークによって高速化の度合いが異なるのは、入力サイズが異なるためです。この実験結果は、インテルのハイパー・スレッディング・テクノロジが初めて公開された 2001 Microprocessor Forum で披露されたものです [2]。

ベンチマーク	説明	高速化
Synthetic	大規模なデータベース検索をシミュレートするランダム・グラフにおける、グラフ・トラバース	22～45%
MST (Olden)	データ・クラスタリングに利用される Minimal Spanning Tree アルゴリズム	23～40%
Health (Olden)	医療用システムをモデル化した階層データベース	11～24%
MCF (SPEC2000int)	バス・スケジューリングに利用される整数プログラミング・アルゴリズム	7.08%

表 3: 初期のパフォーマンス・データ：製造前段階のハイパー・スレッディング・テクノロジ対応インテル® Xeon™ プロセッサにおける SP

今回の実験で使用したシリコンは、ハイパー・スレッディング・テクノロジの実装製品として第一世代にあたるものです。このチップには 2 つのハードウェア・スレッド・コンテキストが用意されており、ハイパー・スレッディング・テクノロジへの最適化が行われた Microsoft Windows* XP オペレーティング・システムで動作させることができます。2 つのハードウェア・コンテキストは、ユーザからは 2 つの論理プロセッサによる SMP (Symmetric Multiprocessing) として認識されます。オンチップのキャッシュ階層は 2001 年現在発売されているインテル Pentium® 4 プロセッサと同じ構成となっており、このオンチップ・キャッシュ階層の全体を 2 つのハードウェア・スレッドが共有するようになっています。また、このチップには SP をサポートする特別なハードウェアは一切用意されていません。このセクションでは、表 3 に示した Synthetic ベンチマークの擬似コードを用いて、SSP を適用する方法論について解説します。

図 10 はこのマイクロベンチマークの擬似コードを示したものです。図 11 と図 12 は、メイン・スレッド、およびプリフェッチを行う SP スレッドのそれぞれの擬似コードを示したものです。

SP: Main Thread

```
1 main()
  {
2   CreateThread(T
3   WaitForSingleObject

4    n =NodeArray[0
5    while(n and
     {
6 work()
7 n->i = n->next->j + n->next->k + n-
8 n = n-
9 remaining-
10 Every stride
11 global_n =
12 global_r =
13 SetEvent(
     }
  }
```

Line 11-12: Live-in's for cross thread transfer
Line 13: Trigger to activate SP thread

図 10：シングルスレッド・コードの擬似コード、および delinquent ロードのプロファイリング結果

先のセクションで一般的な SP のタスクを説明したように、ここでも delinquent ロードの特定、SP スレッドの作成、SP トリガの埋め込み、メイン・スレッド

とスペキュレーティブ・スレッドとの間で live-in ステートを転送するためのメカニズムが必要となります。

delinquent ロードの特定は、インテル VTune™ パフォーマンス・アナライザ 6.0 [16] を用いて行えます。例えば、図 10 に示したとおり、5 行目のポインタ 2 重参照ロードは L2 キャッシュ・ミスを引き起こす delinquent ロードで、きわめて大きいレイテンシを引き起こします。

子スレッドの生成を明示的にサポートするハードウェアは利用しないため、スレッド間での通信および状態転送を行うために、ここでは標準の Win32* スレッド API を使用しています。`CreateThread()` は初期設定時に SP スレッドを生成するために使います。`SetEvent()` は基本トリガをメイン・スレッド内に埋め込むのに使用します。`WaitForSingleObject()` は SP プリフェッチ・スレッド内で使用し、対応するスペキュレーティブ・スレッドのイベント駆動型アクティベーションを実装します。さらに、スレッド間で状態を明示的に転送するための媒体としてグローバル変数を使用しています。メイン・スレッドでは、`SetEvent()` でトリガ・イベントを送出する前に live-in バリューをこのグローバル変数に格納するようにしています。一方、SP プリフェッチ・スレッドではポインタ追跡プリフェッチを実行する前にグローバル変数から live-in バリューを読み込むようになっています。

SP: Main Thread

```
1 main()
  {
2   CreateThread(T
3   WaitForSingleObject
    ()
4   n =NodeArray[0
5   while(n and
    {
6 work()
7 n->i = n->next->j + n->next->k + n-
8 n = n-
9 remaining-
10 Every stride
11 global_n =
12 global_r =
13 SetEvent(
    }
  }
```

Line 11-12: Live-in's for cross thread transfer
Line 13: Trigger to activate SP thread

図 11：SP におけるメイン・スレッドの擬似コード

SP: Worker Thread

```
1 T()
  {
2  Do Stride times
3     n->i = n->next->j + n->next->k + n->next->l
4     n = n->next
5     remaining--
6  SetEvent()
7  while(n and remaining)
   {
8    Do Stride times
9       n->i = n->next->j + n->next->k + n->next->l
10      n = n->next
11      remaining--
12   WaitForSingleObject()
13   if (remaining < global_r)
14 remaining = global_r
15 n = global_n
   }
  }
```

Line 9: Responsible for Most effectiveprefetchdue to run-ahead
Line 13: Detect run-behind, adjust by jumping ahead

図 12：SP でプリフェッチを行うスペキュレーティブ・スレッドの擬似コード

また、図 12 に示したとおり、プリフェッチを行う SP スレッドには、SP の動作を調整するためのシンプルかつきわめて重要なメカニズムが採用されています。このメカニズムは、SP スレッドが次の 2 つの重要なステップを確実に行えるようにしています。

1. SP スレッドがアクティブになったら、ポインタを追跡する一連の「ストライド」イタレーションをメイン・スレッドとは独立して必ず実行する。

   「ストライド」で境界づけられたポインタ追跡ループが、効果的に連鎖トリガ機構を認識することに注意してください。処理は複数のイタレーションにわたって行われ、メイン・スレッドの進行状況には依存しません。

2. 「ストライド」イタレーションを 1 回終了するたびに、メイン・スレッドの進行状況を確認し、実行が遅れていないことを確認する。

   メイン・スレッドよりも実行が遅れている場合は、SP スレッドはグローバル・ポインタを同期化させることによってメイン・スレッドに進行を合わせます。

   さらに、SP スレッドの実行が進みすぎないように調整を行うためのコードも使用されています。メイン・スレッドおよび SP スレッドにあるスレッド・ローカル変数「`remaining`」は、それぞれの進行状況を記録するカウンタの役目を果たします。

ここで興味深いのは、SP スレッドは一般的なロード命令しか使用しておらず、いわゆるプリフェッチ命令は一切使用していないにもかかわらず、メイン・スレッドに対して効果的なプリフェッチを実現するという点です。

パフォーマンスを公正に比較するため、ここでは Win32 API ルーチンの `timeGetTime()` を使って、オリジナル・コードと SSP 対応コードの絶対的な実行時間 (ウォール・クロック) を計測し、比較を行っています。なお、これらのコードはいずれもインテル IA-32 C/C++ コンパイラ [35] でビルドされており、最高の速度が得られるよう最適化されています。例示したマイクロベンチマークについて、SSP 対応コードの方がなぜ高速に実行できるのか、その理由を VTune パフォーマンス・アナライザ 6.0 [16] から得たプロファイリング情報を使って説明したのが図 13 です。その内容を要約すると、SP スレッドでは特定した delinquent ロードについてキャッシュ・ミスのほとんどをプリフェッチすることに成功しています。この最適化によって、各種入力サイズで 22% ～45% の高速化が実現しています。

Main Thread
•*Line 7* corresponds to *Line 5* of single thread code
○Execution time:
19% vs 49.46% in single-thread code
○L2 miss:
0.61% vs 99.95% in single-thread code
SP worker thread:
•*Line 9*.
○Execution time:
26.21%
○L2 miss:
97.61%

SP successful in shouldering most L2 cache misses

図 13：SSP 対応コードが高速実行できる理由

もちろん、この実験が成功したことは、これまでのシミュレーション・ベースの研究で得られた SP のアイデアや利点が正しいものであったことが実証されたという意味を持ちます。しかしそれ以上に、マルチスレッディング・プロセッサ・リソースを効果的に利用する新しい方法、すなわちシングルスレッド・アプリケーション内で擬似的に「スレッド・レベルの並列化 (TLP)」を利用したり、マルチスレッディング・ハードウェアを使ってレイテンシを削減するといったことが実現可能であることを実証できたという点でも大きな意義があります。

関連研究

スペキュレーティブ・スレッドを利用してキャッシュ・プリフェッチを行おうという初期の試みとしては、Chappel 他による『Simultaneous Subordinate Microthreading (SSMT)』[27]、Sundaramoorthy 他による『Slipstream Processors』[28]、Song 他による『Assisted Execution』[29] などの研究があります。

SP に関するわれわれの研究 [9][10][11][12][13][14] 以外にも、最近になってスレッド・ベースのプリフェッチ・パラダイムがいくつか提唱されています。Zilles と Sohi による『Speculative Slices』[21]、Roth と Sohi による『Data Driven Multithreading』[22]、Luk による『Software Controlled Pre-Execution』[23]、Annavaram 他による『Data Graph Precomputation』[24]、Moshovos 他による『Slice-Processors』[25] などです。これらの手法のほとんどは、基本トリガによる SP メカニズムと同等のものです。

Roth ほかの指摘にもあるように [30]、これらのスレッド・ベースのプリフェッチによるアプローチはアクセスと実行を論理的に分離したものの一種といえます。これは最初に Smith が考案したもので [31]、その後参考資料 [32][33][34] で研究が進められています。ここでは、専用のメモリ・アクセス・エンジンを物理的に分離してしまうのではなく、アクセス機能はプリフェッチを行う SP スレッドが実行します。post-pass SSP ツールを使うと、特別な「アクセス」スレッドが本来のコードに追加されます。「アクセス」スレッドと「実行」スレッドは、汎用の SMT または CMP プロセッサの別々のハードウェア・スレッド・コンテキスト上でオーバーラップして実行 (パイプライン化) されます。

今回の研究が他と異なる点としては、連鎖トリガのメカニズムを発見したこと、SP と OOO という 2 つの異なるメモリ・レイテンシ隠蔽手法のトレードオフを詳細に分析したこと、完全に自動化された post-pass コンパイル・ツールを用いてバイナリの SSP 化に成功したこと、ハイパー・スレッディング・テクノロジ対応の実際のハードウェアを用いて実験を行い、SSP を利用することでシングルスレッド・ベンチマークの大幅な高速化に成功したことなどが挙げられます。

まとめ

本稿では、スペキュレーティブ・プリコンピュテーション (SP) に関するインテルの研究の主な成果を紹介してきました。SP とは、マルチスレッド・プロセッサ上で有効に利用されていないハードウェア・コンテキストを使ってスペキュレーティブ・スレッドを生成し、メイン・スレッドで必要となるデータを早い段階でプリフェッチしておくという手法です。基本的に、今回の研究では SMT (Simultaneous Multithreading) プロセッサのリソースを効果的に使うことでレイテンシを削減し、シングルスレッド・アプリケーションのパフォーマンスを改善できることが実証されました。

マルチタスク、マルチスレッドのワークロード環境があれば、数多くのスレッドが同時に SMT プロセッサ上で実行されるため、スループットは向上します。しかし SP はこれとは方向性を異にし、シングルスレッド・アプリケーションから補助的なスレッドを抽出し、レイテンシを削減することを目標としています。つまり、SP とは SMT プロセッサにおける命令実行のスループットを高めるのではなく、キャッシュ・ミスのレイテンシ (これはクロック周波数の向上とともに深刻化します) を削減することによってパフォーマンスの改善を図るものであるといえます。SP スレッドでごくわずかの命令を実行するだけで、SP スレッドの実行そのものに必要なレイテンシをはるかにしのぐレイテンシを削減することができます。従来、アプリケーションをマルチスレッド化しようとしても、スレッド・コンテキストを増やして同時に実行できる命令には限界があり、このことが処理速度の向上を阻んでいました。

参考資料 [2] でも説明しているように、インテル® Xeon™ プロセッサで初めて導入されたハイパー・スレッディング・テクノロジの登場は、命令レベルの並列化 (ILP) からスレッド・レベルの並列化 (TLP) への移行という、新たな時代の到来を意味しています。マルチスレッディング手法は、将来のマイクロアーキテクチャ設計において消費電力と複雑さを抑えるという意味でも有効です。インテルでは今後もさらに効果的なマルチスレッディング・リソースの利用法について研究を進めていく考えです。一言でまとめると、スペキュレーティブ・プリコンピュテーション (SP) とは、本来スレッド・レベルの並列化 (TLP) のために用意されているリソースを活用してメモリ・レベルの並列化 (MLP) を高めるというものです。これにより、従来のシングルスレッド・アプリケーションの命令レベルの並列化 (ILP) がいっそう高まることになります。

謝辞

スペキュレーティブ・プリコンピュテーションの研究チームに対して、Justin Rattner (インテル・フェロー。インテル・ラボ、マイクロプロセッサ・リサーチ (旧 MRL) ディレクタ)、Richard Wirt (インテル・フェロー。ソフトウェア・ソリューション事業本部 (SSG) ジェネラル・マネージャ) の両氏から多大なるご支援を賜りました。このほか、各方面から貴重な助言をいただいた次の各氏に対して感謝の意を表します。Kevin J. Smith、David Sehr、Wilfred Pinfold、Jesse Fang、George K. Chen、Gerolf Hoflehner、Dan Lavery、Wei Li、Xinmin Tian、Sanjiv Shah、Ernesto Su、Paul Grey、Ralph Kling、Jim Dundas、Wen-hann Wang、Yong-fong Lee、Ed Grochowski、Gadi Ziv、Shalom Goldenberg、Shai Satt、Ido Shamir、Per Hammarlund、Debbie Marr、John Pieper、Bo Huang、Young Wang、Rob Portman、Pete Andrews、Tony Martinez、Oren Gershon、Jonathan Beimel、Ady Tal、Yigal Zemach、Leonid Baraz。

今回の SP に関する研究においては、カリフォルニア大学サンディエゴ校の学部長 M. Tullsen 教授および研究チームの方々と終始密接なコラボレーションを行うことができたことを感謝いたします。特に、Tullsen 教授のもとで博士論文に取り組んでおられた Jamison D Collins 氏から数々の重要な貢献をいただくことができました。

最後に、本稿に対し貴重な助言をいただいた論文審査員の方々にも感謝いたします。さらに、丹念に編集を行っていただいた Lin Chao、Judith Anthony、Marian Lacey の 3 名にも感謝の意を表します。

参考資料

[1] J. Emer 著『Simultaneous Multithreading: Multiplying Alpha's Performance』、Microprocessor Forum、1999 年 10 月

[2] G. Hinton、J. Shen 著『Intel's Multi-Threading Technology』、Microprocessor Forum、2001 年 10 月

[3] D. M. Tullsen、S. J. Eggers、H. M. Levy 著『Simultaneous Multithreading: Maximizing On-Chip Parallelism』、22nd ISCA、1995 年 6 月

[4] D. M. Tullsen、J. A. Brown 著『Handling Long-Latency Loads in a Simultaneous Multithreading Processor』、Micro-34、2001 年 12 月、pp. 318-327

[5] T. Chen 著『An Effective Programmable Prefetch Engine for On-chip Caches』、Micro-28、1995 年 12 月、pp. 237-242

[6] N. Jouppi 著『Improving Direct-mapped Cache Performance by the Addition of a Small Fully associative Cache and Prefetch Buffers』、ISCA-17、1990 年 5 月、pp. 364-373

[7] D. Joseph、D. Grunwald 著『Prefetching using Markov Predictors』、ISCA-24、1997 年 6 月、pp. 252-263

[8] T. Mowry、A. Gupta 著『Tolerating Latency through Software-controlled Prefetching in Shared-memory Multiprocessors』、Journal of Parallel and Distributed Computing、1991 年 6 月、pp. 87-106

[9] H. Wang, et al., "Software-based Speculative Precomputation and Multithreading," in *Docket No. 042390.P10811*, Patent Pending, March 2001.

[10] H. Wang、et al. 著『Software-based Speculative Precomputation and Multithreadings』、Docket No. 042390.P10811、Patent Pending、2001 年 3 月

[11] J. Collins、H. Wang、D. Tullsen、C, Hughes、Y-F Lee、D. Lavery、J. Shen 著『Speculative Precomputation: Long-range Prefetching of Delinquent Loads』、28th ISCA、2001 年 7 月

[12] J. Collins、D. Tullsen、H. Wang、J. Shen 著『Dynamic Speculative Precomputation』、Micro-34、2001 年 12 月、pp. 306-317

[13] P. Wang、H. Wang、J. Collins、E. Grochowski、R. Kling、J. Shen 著『Memory latency-tolerance approaches for Itanium processors: out-of-order execution vs. speculative precomputation』、Proceedings of the 8th IEEE HPCA、2002 年 2 月

[14] S. S. W. Liao、P. Wang、H. Wang、G. Hoflehner、D. Lavery,、J. P. Shen 著『Post-pass Binary Adaptation for Software-based Speculative Precomputation』、Accepted for publication at PLDI'02

[15] S. G. Abraham、B. R. Rau 著『Predicting Load Latencies using Cache Profiling』、HP Lab Technical Report HPL-94-110、1994年12月

[16] インテル コーポレーション、VTune パフォーマンス・アナライザ http://www.intel.co.jp/jp/developer/software/products/VTune/index.htm

[17] C. Zilles、G. Sohi 著『Understanding the backward slices of performance degrading instructions』、27th ISCA、2000年6月、pp. 172-181

[18] D. P. Bhandarkar、Alpha Implementations and Architecture、Digital Press, Newton, MA、1996年

[19] J. Heinrich、MIPS R10000 Microprocessor User's Manual、MIPS Technologies Inc.、1996年9月

[20] G. Hinton、D. Sager、M. Upton、D. Boggs、D. Carmean、A. Kyker、P. Roussel『Pentium® 4 プロセッサのマイクロアーキテクチャ』インテル・テクノロジ・ジャーナル、2001年第1四半期号 http://www.intel.co.jp/jp/developer/technology/itj/q12001/articles/art_2.htm

[21] C. Zilles、G. Sohi 著『Execution-based prediction using speculative slices』、28th ISCA、2001年7月

[22] A. Roth、G. Sohi 著『Speculative Data-Driven Multithreading』、7th HPCA、2001年1月

[23] C. K. Luk 著『Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors』、28th ISCA、2001年6月

[24] M. Annavaram、J. Patel、E. Davidson 著『Data Prefetching by Dependence Graph Precomputation』、ISCA-28、2001年7月、pp. 52-61

[25] A. Moshovos、D. Pnevmatikatos、A. Baniasadi 著『Slice processors: an implementation of operation-based prediction』、International Conference on Supercomputing、2001年6月

[26] A. Roth、A. Moshovos,、G. Sohi 著『Dependence-based prefetching for linked data structures』、ASPLOS-98、1998年10月、pp. 115-126

[27] R. Chappell、J. Stark、S. Kim、S. Reinhardt、Y. Patt 著『Simultaneous Subordinate Microthreading (SSMT)』、26th International Symposium on Computer Architecture、1999年5月

[28] K. Sundaramoorthy、Z. Purser、E. Rotenberg 著『Slipstream Processors: Improving both Performance and Fault Tolerance』、9th ASPLOS、2001年11月

[29] Y. Song、M. Dubois、Assisted Execution. Technical Report #CENG 98-25 Department of EE-Systems、University of Southern California、1998年10月

[30] A. Roth、C. B. Zilles、G. S. Sohi 著『Microarchitectural Miss/Execute Decoupling』、MEDEA Workshop、2000年10月

[31] J. E. Smith 著『Decoupled Access/Execute Computer Architecture』、9th ISCA、1982年7月

[32] M. K. Farrens、P. Ng、P. Nico 著『A Comparison of Superscalar and Decoupled Access/Execute Architectures』、26th Micro、1993年11月

[33] G. P. Jones、N. P. Topham 著『A Limitation Study into Access Decoupling』、3rd Euro-Par Conference、1997年8月

[34] J. M. Parcerisa、A. Gonzalez 著『The Synergy of Multithreading and Access/Execute Decoupling』、5th HPCA、1999年1月

[35] A. Bik、M. Girkar、P. Grey、X. Tian 著『Pentium® III および Pentium® 4 プロセッサ・ベースのシステムによる並列実行性の効率的実現』インテル・テクノロジ・ジャーナル、2001年第1四半期号 http://www.intel.co.jp/jp/developer/technology/itj/q12001/articles/art_6.htm

著者紹介

Hong Wang、インテル・ラボ、マイクロプロセッサ・リサーチ所属の上級スタッフ・エンジニア。独創的なアイデアを発見し、将来のインテル・プロセッサの設計に応用することを目標に研究に取り組む。1995年、インテルに入社。1996年、ロードアイランド大学にて電気工学の博士号を取得。

電子メールアドレス：hong.wang@intel.com

Perry H. Wang、インテル・ラボ、マイクロプロセッサ・リサーチ事業本部に所属。1995年、インテルに入社。先進マイクロアーキテクチャ、およびコンパイラの最適化を専門とする。ミシガン大学アン・アーバー校にて物理工学の学士号およびコンピュータ・サイエンスの修士号を取得。

電子メールアドレス：perry.wang@intel.com

R. David Weldon、2000年、インテルに入社。現在は、ロジック・テクノロジ開発事業本部にて将来の IA-32 プロセッサの設計に携わる。ワシントン大学にて電気工学の学士号、およびコーネル大学にて修士号を取得。プロセッサのマイクロアーキテクチャ、およびハードウェア/ソフトウェアのコデザインを専門とする。
電子メールアドレス：ross.d.weldon@intel.com

Scott M. Ettinger、2001年、インテルに入社。コンピュータ・アーキテクチャおよびマルチメディア信号

処理を専門とする。フロリダ大学にて電気工学の学士号と修士号を取得。

電子メールアドレス：scott.m.ettinger@intel.com

Hideki Saito、1993年に京都大学にて情報科学の学士号、1998年に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・サイエンスの修士号を取得。現在、同校にて博士論文提出済み。2000年6月、インテル コーポレーションに入社以来、マルチスレッディングおよびパフォーマンス解析に取り組む。現在、OpenMP Parallelization グループにもメンバーとして参加。

電子メールアドレス：hideki.saito@intel.com

Milind Girkar、コンピュータ・サイエンスを専攻し、インド工科大学ムンバイ校にて学士号、バンダービルト大学にて修士号、イリノイ大学アーバナ・シャンペーン校にて博士号をそれぞれ取得。現在はインテルのソフトウェア・ソリューション事業本部にて IA-32 コンパイラ開発チームのマネージャを務める。インテル入社前は、Sun Microsystems において UltraSPARC プラットフォーム用コンパイラの開発に従事。

電子メールアドレス：milind.girkar@intel.com

Steve Shih-wei Liao、インテル・ラボ、マイクロプロセッサ・リサーチ事業本部所属。国立台湾大学にてコンピュータ・サイエンスの学士号、スタンフォード大学にて電気工学の修士号と博士号を取得。プログラム解析/最適化、コンピュータ・アーキテクチャ、プログラミング環境を専門とする。

電子メールアドレス：shih-wei.liao@intel.com

John P. Shen、インテル・ラボ、マイクロアーキテクチャ・リサーチのディレクタ。2000年、インテルに入社。それ以前はカーネギー・メロン大学で 18年以上電気工学/コンピュータ工学部で教鞭を取る。現在は IEEE フェローでもあり、現在は「Fundamentals of Superscalar Processor Design」(2002年、McGraw-Hill より刊行予定) のテキストブックを執筆中。

電子メールアドレス：john.shen@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 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.