インテル® Itanium® 2 プロセッサ マイクロアーキテクチャ最適化入門

リファレンス・マニュアル

インテル® Itanium® 2 プロセッサ マイクロアーキテクチャ最適化入門

リファレンス・マニュアル

資料番号 : 251464-001J

Web : http://www.intel.co.jp/jp/developer/ (日本語)
: http://developer.intel.com (英語)

バージョン	バージョン情報	日付
-0	初版	07/ 2002 年 7 月

目次

はじめに

1

本書はソフトウェア開発者を対象に、Itanium® 2 プロセッサのパフォーマンス監視イベントを体系的に使用して VTune™ パフォーマンス・アナライザでアプリケーションの実行効率を分析する方法について説明する。パフォーマンス・カウンタ、ソフトウェア最適化、VTune アナライザについては、それぞれ専門に解説したリファレンス・ドキュメントがインテルから提供されている（本章最後の参考文献を参照）。本書では、パフォーマンス・カウンタ、VTune アナライザ、コンパイラを利用したマイクロアーキテクチャ最適化を総合的な視点から解説する。本書は以下に紹介する参考文献の代わりとなるものではなく、これら文献の入門的な役割を果たすものである。

本書の対象読者は、パフォーマンスに対する要求の厳しいアプリケーション、ソフトウェア開発ツール、デバイス・ドライバ、オペレーティング・システムを開発しているソフトウェア・デベロッパである。Itanium プロセッサ・ファミリ・アーキテクチャでパフォーマンスを最大化するには、明示的なスケジューリングを行う必要がある。したがって、Itanium プロセッサ・ファミリ用にソフトウェア開発を行う際は、高級言語でコードを開発し、最新バージョンのコンパイラを使うことが不可欠である。本書では主に、高級言語で開発されたコードの最適化方法について解説する。

Itanium 2 プロセッサに最適化したコードは、今後の新しい世代の Itanium プロセッサに対しても有効である。その場合、新しいプロセッサのアーキテクチャ上の変更に対応するための再コンパイルさえ行えば対応できる。なお、本書で紹介するマイクロアーキテクチャ最適化の手法は、アセンブラでハンド・コーディングする際にも通用するが、コンパイラによる再スケジューリングによって将来のアーキテクチャに対応できないため、アセンブラの使用は避けるべきである。

アルゴリズムとデータ構造がマイクロアーキテクチャに最適化されていないと、データの読み出し時および書き込み時に実行効率が低下する。現在のところ、本書では読み出し時の実行効率の改善についてのみ解説している。データ書き込み時の実行効率を改善する方法については、今後の改訂時に取り上げる予定である。

参考文献

1.『インテル® Itanium® 2 プロセッサ・リファレンス・マニュアル：ソフトウェアの開発と最適化』（Revision 1.0）

2.『Itanium® Architecture for Software Developers』（W. Triebel 著、インテル®・プレス刊）

3. VTune™ パフォーマンス・アナライザのオンライン・ヘルプ

4.『Software Optimization for High Performance Computing』（K.R Wadleigh、I. L. Crawford 著、Hewlett-Packard* Professional Books 刊）

5. Itanium® 2 プロセッサのマイクロアーキテクチャ仕様

6.『インテル® Itanium® 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』（第 1 巻～第 3 巻）

マイクロアーキテクチャ最適化の概要

2

マイクロアーキテクチャへの最適化を行ってパフォーマンスをチューニングするのは、命令がプロセッサの機能ユニットに効率よく流れるようにして、実行効率を最大化することである。このため、本書では以下の事項についてはすでに実施済みであることを前提として解説を進める。

- アルゴリズムのコーディングが適切に行われている
- スペースとスピードのトレードオフが最適な形で行われている
- 十分なメモリが搭載されており、アプリケーションから余分なディスク・アクセスが発生しない
- 必要な周辺機器（ネットワーク・インターフェイス、ビデオなど）に十分な帯域幅があり、パフォーマンスのボトルネックとなっていない

また、パフォーマンス分析を行うマシンが適切に構成されていることも前提条件となる。

パフォーマンス監視カウンタとパフォーマンス分析ツールを使ってカウンタ・データを収集すれば、機能ユニットへの命令フローのボトルネックとなっている部分を特定できる。この種のツールとして最も広く使われているものに、VTune™ パフォーマンス・アナライザがある。VTune アナライザには数多くの機能が用意されている。ただし、VTune アナライザは例外ハンドラを利用してカウンタ・データを収集しており、通常のプロセッサ実行に対してきわめて大きなオーバーヘッドを発生する。一般的にはグローバルな情報だけで分析が行えるケースも多いため、こうした場合には Emon などオーバーヘッドの少ないツールでデータを収集すれば十分である。

いずれにしても、パフォーマンス分析ツールはマシン全体のサンプリングを行うため、有効なデータを得るにはマシンをサンプリング専用に使用しなければならない。データ収集中に別のユーザなどが他のタスクを実行すると、OS が複数のプロセス間でマルチタスクを行うと正しいデータ収集が行えなくなるので注意が必要である。

パフォーマンス・カウンタ・データを適切に収集するには、プロセッサのマイクロアーキテクチャを十分に理解しておく必要がある。パフォーマンス・カウンタは、アプリケーションが実行されている間、プロセッサのアーキテクチャ機能の利用状況や呼び出し状況を記録する。この情報をもとに、アプリケーションがマイクロアーキテクチャを効率的に利用できていない部分を特定できる。パフォーマンス・カウンタを効果的に使ってアプリケーションの実行効率を分析するには、さまざまな種類のパフォーマンス・カウンタ、測定対象となるアーキテクチャの細かい機能を十分に理解しておく必要がある。

本書では、マイクロアーキテクチャ、パフォーマンス分析ツール、そしてパフォーマンス・カウンタの体系的な使用法について解説する。これら個々の内容については、専門に解説したドキュメントがあるのでそちらを参照のこと（『インテル® Itanium® 2 プロセッサ・リファレンス・マニュアル：ソフトウェアの開発と最適化』Revision 1.0）。

本書では、以下の点を重点的に取り上げる。

- アプリケーションの実行効率低下の原因をマイクロアーキテクチャ面から解説。特に、パフォーマンス・カウンタを利用してプロセッサの動作を監視し、マイクロアーキテクチャ上のパフォーマンス・ボトルネックを特定する作業について取り上げる。
- VTune パフォーマンス・アナライザについての解説。アプリケーションを実行させてパフォーマンス監視データを収集する機能について説明する。

パフォーマンス・モニタを体系的に使用するという方法論は、Itanium プロセッサ・ファミリのサイクル・アカウンティング機能を利用して生まれたものである。Itanium 2 プロセッサでは、サイクル・アカウンティング・イベントを使用して実行効率低下の主な原因を調べる方法論が定義されている。本書は主にこの分析手法について解説を行い、さらにソース・コードやコンパイルの変更により実行効率を高めていくための一般的な方法を紹介する。

Itanium® 2 プロセッサのアーキテクチャ

3

コードを最適化してマイクロプロセッサのアーキテクチャ機能を最大限に利用するには、対象となるアーキテクチャの理解が不可欠である。本書で解説するマイクロアーキテクチャ最適化とは、高級言語で記述したコンパイル済みコードをマイクロプロセッサが効率よく読み込んで実行できるようにすることである。ソフトウェア開発者はこのプロセスに消極的であってはならない。データやアルゴリズムの構造を積極的に修正していくことによって、アーキテクチャの持つ機能を最大限に活用し、単純な見落としによる実行効率の低下を避けねばならない。

本書では、EPIC（明示的並列命令コンピューティング）アーキテクチャの基礎については扱わない。これについては、他のドキュメントを参照されたい（『インテル® Itanium 2 プロセッサ・リファレンス・マニュアル：ソフトウェアの開発と最適化』Revision 1.0 http://developer.intel.com/design/itanium/manuals.htm）。本書では、Itanium 2 プロセッサのアーキテクチャの中でも、ソフトウェア・パフォーマンスのチューニングを行う際に必要となる部分に限って説明を行う。読者には、Itanium プロセッサ・ファミリ・アーキテクチャおよびアセンブラに関する相応の知識があることを前提とする。

マイクロプロセッサがアプリケーションを実行するということは、命令および必要なデータを同期化させて実行ユニットに送り込むということである。この流れがスムーズであれば、プロセッサが命令やデータを待つストールが発生せず、実行効率は最適となる。本章ではこのように命令とデータを同期化させて実行ユニットに送り込む方法をアーキテクチャ面から解説していく。

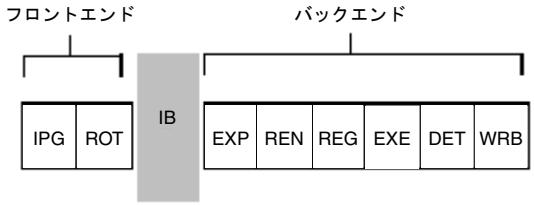
命令とデータを機能ユニットに送り込む作業はプロセッサのコア・パイプラインが制御しており、この部分を最適化することがチューニング作業の中心となる。パイプラインの最も上流では、パイプラインのバックエンドで処理しやすいようにアライメントされた命令が命令キャッシュまたは ISB（Instruction Streaming Buffer）から読み込まれる。バックエンドでは、まずバンドルのテンプレートを展開し、命令を機能ユニットに配布する。次に、レジスタのリネームが開始され、レジスタから機能ユニットにデータが送られる。こうしてデータと命令が揃ったら、命令が実行される。次に、分岐予測ミスや例外のチェックを行う例外処理ステージがあり、最後に、要求された結果をレジスタに書き戻す。

以上のプロセスは、パフォーマンス・カウンタを分析すれば細部まで詳しく把握できる。このようにしてアプリケーションを最適化し、パフォーマンスを最大化していく。

コード・フローの制約とコア・パイプライン

コア・パイプラインは命令とデータを同期的に機能ユニットに発行できるように準備を整える。8 段のパイプラインは 2 段のフロントエンドと 6 段のバックエンドの2つで構成され、これらは命令バッファ（IB: Instruciton Buffer）を間にはさんで相互に接続されている（下図参照）。パイプラインのフロントエンドとバックエンドは互いに非同期的に動作する。

図 3-1　コア・パイプライン



フロントエンドではまず命令ポインタを生成し（IPG ステージ）、L1-I キャッシュおよび L1-ITLB へのアクセスを開始する。次に命令ストリームをフォーマットし（ROT ステージ）、命令バッファへのロードが完了したら、バックエンドに処理が渡される。

バックエンドは全体で 6 ステージ構成となっており、命令バッファから命令を受け取り、必要なデータを適切なレジスタに割り当て、機能ユニットに命令を発行する。

ステージ 1（EXP）

最初の EXP ステージでは命令テンプレートを展開し、命令のディスパーサルを準備、実行する。

ステージ 2（REN）

REN ステージはレジスタ・スタックに対してレジスタのリネームを行うとともに、レジスタのローテーションを行う。また、この命令のデコードもこのステージで行う。

ステージ 3（REG）

REG ステージでは、機能ユニットに対してデータの配布が行われる。データの配布はレジスタから行われる場合と、機能ユニットによって生成されたデータを次のチェーン命令に引き渡すバイパスを利用して行われる場合の 2 通りがある。また、レジスタ・スタック・エンジン（RSE）から要求されたスピルまたはフィル命令の生成もこのステージで行う。

ステージ 4（EXE）

EXE ステージでは、命令テンプレートで指定された機能ユニットに対して命令とデータをディスパッチする。また、レイテンシ 1 サイクルの ALU 命令からの出力データを REG ステージに渡して後続の命令で使用できるようにするバイパスもここで呼び出される。

ステージ 5（DET）

DET ステージでは例外および分岐予測ミスの検出を行う。例外や分岐予測ミスによるパイプライン・フラッシュが発生するのはこのステージであり、ここで発生するストールがパイプライン中で最も優先度が高い。潜在的な例外はすべてこのステージで検出され、アーキテクチャ・ステートのライトバックを防ぐようにしている。こうすれば、アーキテクチャ・ステート（レジスタの内容など）を正しい状態に維持できる。コア・パイプラインのストールの原因となるデータ配布のストールや浮動小数点マイクロパイプラインのストールについても、このステージで検出される。

ステージ 6（WRB）

最後にこのライトバック（WRB）ステージで、結果が適切な出力レジスタに書き戻される。

パイプラインのバックエンドにおけるストール

本書で解説する最適化方法論は、パイプラインのバックエンドにおけるプロセスを最適化し、機能ユニットに対して命令を効率よく配布することに主眼を置いている。つまり、パイプラインのバックエンドで発生するストール・サイクルを最小限に抑えるのが目的である。パイプラインのバックエンドには、ストールを発生させる可能性のあるステージが 3 つある。コア・パイプラインのフロントエンドとバックエンドは非同期的に動作しているため、フロントエンドのストールについては、バックエンドのストールにつながるものしか論じない。また、パイプラインとマイクロパイプラインの相互作用にも目を向けると、パイプラインのバックエンドで発生するストールは大きく 5 種類に分類できる。

バックエンド・パイプラインのストール状態には優先度があり、それはパイプラインの終わりに近づくにつれて高くなる。最も優先度が高いのは、DET ステージで例外や分岐予測ミスが検出されて発生するストールである。この場合、パイプラインがフラッシュされ、その時点でパイプラインの上流で進められていた処理はすべて無効になる。

パイプラインのバックエンドは FPU、マルチメディア・ユニット、L1-D および L2 キャッシュ・アクセスなどを制御しているいくつかのマイクロパイプラインと連携している。こうした FPU や L1-D マイクロパイプラインとの連携が原因で、パイプラインのバックエンドの DET ステージでストールが発生する場合もある。ただし、このようなストールは例外や分岐予測ミスによるストールよりも優先度は低い。

アプリケーション実行中に EXE ステージで発生する主なストールは、依存性スコアボードによるストールである。これは機能ユニットが必要とするデータが事前にレジスタに到着していない場合に発生する。正しい結果を得るために、パイプラインはデータがレジスタに到着するまでの間、ストールする。データの到着が遅れるケースとしては、命令のレイテンシが大きい場合（浮動小数点やマルチメディア）や、ロードがスケジュールされたサイクル数の範囲内で完了しなかった場合などがある。

REN ステージでパイプライン・ストールが発生するのは、汎用レジスタに十分な空きがなく、処理すべき alloc 命令を 1 つも実行できない場合である。この場合、REG ステージが必要なスピル命令またはフィル命令を生成して、レジスタ・スタック・エンジンを呼び出す。

最後に、フロントエンドがバックエンドに対して実行フローの継続に十分な命令フローを供給できない場合にも、バックエンドは命令待ちのためにサイクルを消費し、プログラム実行が停滞してしまう。

メモリ・サブシステム

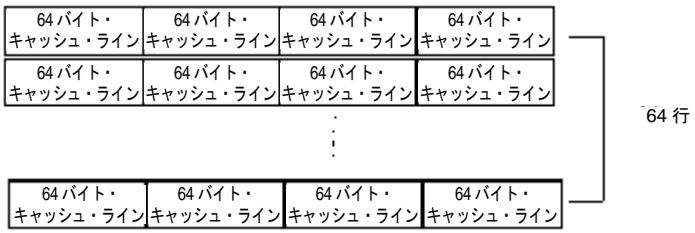
チューニング作業の大半は、コードとメモリ・サブシステム間の双方向のやりとりを最適化した後、アルゴリズムやデータ構造を変更する作業となる。Itanium® 2 プロセッサは 3 階層のキャッシュを内蔵しており、メイン・メモリのアクセスにはインテル® プラットフォームの場合、870 チップセットを経由する。メモリ・サブシステムを効果的に使用するには、コードとメモリ・サブシステム間のやりとりを最適化することが必要であるため、本書でも主にこの部分について重点的に説明する。

最近のコンピュータの多くはメモリ・キャッシュ・システムを採用しており、頻繁に使用するデータや命令にはきわめて高速なアクセスが可能となっている。データ（および命令）は、メモリ内の連続した部分を格納したキャッシュ・ラインにロードされる。まず最初にメモリに対するアクセス要求（ロード命令）があると、要求されたデータだけでなく、キャッシュ・ライン分の連続したデータがメイン・メモリから読み出される。これは、プログラムがあるデータを要求した場合、その近接するアドレスのデータも要求される可能性が高い事実を前提としている。また、メモリへの書き込みを行う際は、キャッシュ内のデータも書き換えられる。これは、書き込んだデータをその後の命令で使用する場合に正しい値にするためである。特にマルチプロセッサ・システムの場合は、複数のプロセッサのキャッシュ・システムのコヒーレンシを確保し、データの一貫性を維持するためにハードウェア設計に大きな労力が払われる。

キャッシュ・メモリは行と列からなるテーブルとして編成されており、キャッシュ・ラインに格納されたデータが行を構成する。特定のデータ（および関連するキャッシュ・ライン中のデータ）がどの行に格納されるかは、アドレスの一部のビット列で決定される。例えばキャッシュ・ラインのサイズが 64 バイトのシステムでは、アドレスの下位 6 ビットを切り捨てて、どの行に格納するかを決定する。こうして、連続する 64 バイト分のデータ列が隣接する行に格納される。一方、テーブルの列数はキャッシュの「アソシアティビティ」と呼ばれる。キャッシュにはすべてのデータを格納するほどの容量はないため、新しいデータへの要求があるとキャッシュ・ラインは強制的に入れ替えられる。キャッシュ・システムをより効率的に使用するため、キャッシュ・ラインはアソシアティブ・セットにグループ化され、テーブルの各行はこのセット数で分割される。新しいキャッシュ・ラインを読み込む場合は、精巧なアルゴリズムを使用して、セット内で今後すぐに使われる可能性の最も低いものから置き換えていく。アソシアティビティを大きくすると、この置き換えが精密に行われるため、キャッシュの利用効率が高まる。

次に示す図は、64 バイト・ラインの 4 ウェイ・アソシアティブ・キャッシュを示したもので、これはちょうど Itanium 2 プロセッサの L1 データ（L1-D）キャッシュに相当する。行数はキャッシュ・サイズ /256 バイトとなるため、Itanium 2 プロセッサの L1-D キャッシュでは 16KB/256B = 64 行となる。行のインデックスは、キャッシュ・ラインのベース・アドレス（すなわち、要求されたデータのアドレスの直下にある 64 バイト境界にアライメントされたアドレス）に基づいて決められる。つまり、行のインデックスはアドレスの 7 番目のビットから開始してすべての行を表現できるだけのビット数を使って計算される。Itanium 2 プロセッサの L1-D キャッシュの場合、アドレス・サイクルは 64*64 = 4096 バイトである。つまり、アドレスが 4096 バイト間隔にあるデータは同一のアソシアティブ・セットに割り当てられるため、セット内でキャッシュ・ラインの入れ替えを引き起こす可能性がある。

図 3-2　Itanium® 2 プロセッサの L1 データ・キャッシュ



キャッシュの構造を考える際に注意しなければならないのは、L2 キャッシュのようにセット・アソシアティブ・テーブルとして構成されると同時にバンク構造としても構成される点である。バンク構造をとるのは、プロセッサ・コアに対するロードおよびストア・ポートを実装しやすくするなど、製造上の理由であることが多い。しかしこの場合、バンク・アクセス競合によってデータ・アクセスの帯域幅を最大限に利用できない場合がある。

ロード操作で要求したデータがキャッシュ内にある場合を「キャッシュ・ヒット」と呼ぶ。このとき、プロセッサのキャッシュ構造と戦略はその目的を果たしたことになり、低レイテンシでのメモリ・アクセスが可能になる。一方、要求したデータがキャッシュ内に存在しない場合を「キャッシュ・ミス」と呼ぶ。通常、パフォーマンスを最大限に高めるにはアルゴリズムやデータ構造を設計する際にキャッシュ構造を考慮に入れる必要がある。これは、キャッシュの使用効率を最大限に高めようという考え方である。これによってキャッシュ・ミスは最小限に抑えられるが、Itanium 2 プロセッサのような多階層のキャッシュ構造の場合は特にキャッシュの細部を理解しておくことが必要となる。

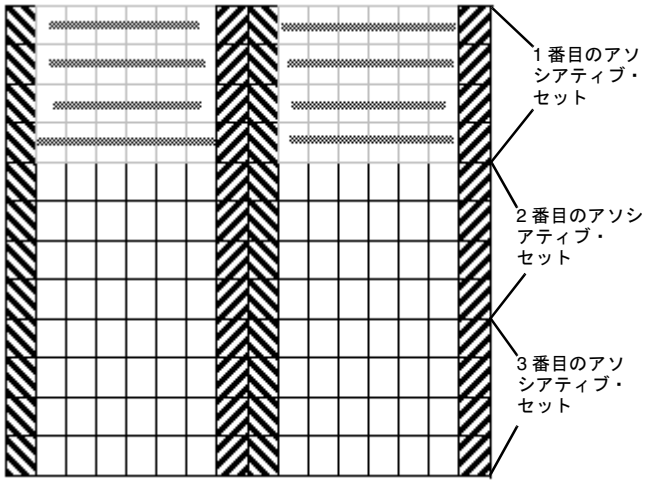
Itanium® 2 プロセッサのキャッシュの説明

メモリ・システムの中で最も高速なのが L1 キャッシュである。L1 キャッシュには命令キャッシュ（L1-I）とデータ・キャッシュ（L1-D）の 2 つがある。Itanium® 2 プロセッサではこれらはいずれもサイズが 16KB で、キャッシュ・ラインは 64 バイト、レイテンシは 1 サイクルである。L1 キャッシュは 4 ウェイ・セット・アソシアティブ・キャッシュであるため、下位 12 ビット（11:0）はキャッシュのインデックスに使用される。

L1 データ・キャッシュには整数データのみを格納する。浮動小数点演算を多用するアプリケーションで必要とされるデータ・サンプルは一般にサイズが大きく、L1 ほどの高速なキャッシュには容量的に格納できないことが研究によって明らかになっている。基本的に、L1 データ・キャッシュへのアクセスにはバンク / ポートの競合は存在しない。L1 データ・キャッシュはライトスルー・キャッシュであり、メイン・メモリには常に最新のデータが格納される。整数ロード（すなわち汎用レジスタのロード）はキャッシュ不可能なメモリからのロードでない限りすべてこの L1 データ・キャッシュを通して行われる。データ・アクセス要求はインオーダーで発行されるが、データはアウトオブオーダーで返される。これをノンブロッキング・キャッシュと呼び、1 回キャッシュ・ミスが発生しても、後続のデータ・アクセスがキャッシュにヒットすればデータを滞りなく読み出せる。L1 データ・キャッシュとレジスタ・ファイルは 2 つのロード用ポートと 2 つのストア・ポートで接続されており、クロック・サイクルに最大2つのロードと2つのストアが可能である。

L2 キャッシュは命令キャッシュとデータ・キャッシュを統合したユニファイド・キャッシュである。L2 キャッシュの容量は 256KB で、アソシアティビティは 8 ウェイ、キャッシュ・ラインのサイズは 128 バイトである。つまり、この L2 キャッシュは合計で 256 個（256KB/8X128 バイト）のアソシアティブ・セットを持つため、図 3-2 のように示せば行数は 256 となる。したがって、この L2 キャッシュのアドレス・サイクルは 32KB である。また、L2 キャッシュは 16 バイト幅のバンク 16 個で構成される。次の図に、16 バンク構成の 8 ウェイ・セット・アソシアティブの L2 ユニファイド・キャッシュの模式図を示す。

図 3-3　キャッシュの物理的構成を示した模式図。アソシアティブ・セットに含まれる各要素（ウェイ）がそれぞれ 16 バイト幅のバンク 8 個で構成されている。



浮動小数点データは L1 キャッシュにはロードされず、L2 キャッシュから浮動小数点レジスタに直接ロードされる。データ・アクセス要求がインオーダーで行われ、データがアウトオブオーダーに利用可能になるのは L1 キャッシュと同じである。これは、FIFO キューを経由して L2 キャッシュにアクセスを行っていた Itanium プロセッサとは異なる。L2 キャッシュの最小レイテンシは、整数ロードで 5 サイクル、浮動小数点ロードで 6 サイクルであるが、これは整数と浮動小数点ではデータをレジスタにロードする経路が異なるためである。また、L2 キャッシュには複雑なキューイングおよびバイパス構造が採用されているため、レイテンシはこの最小値よりも大きくなるときがある。4 つのメモリ・ポートはすべて L2 キャッシュから浮動小数点データをロードするために使用できる。この L2 キャッシュは 16 バイト幅のバンク 16 個で構成されており、場合によってはバンク・アクセス競合が発生する場合がある。バンク競合については、メモリ・アクセスによるストールおよび BE_EXE_Bubble、BE_L1D_FPU_Bubble パフォーマンス・カウンタに関する章で詳しく説明する。

L3 キャッシュもユニファイド・キャッシュである。容量は 1.5MB または 3MB で、前者は 6 ウェイ・セット・アソシアティブ、後者は 12 ウェイ・セット・アソシアティブである。キャッシュ・ラインは 128 バイトで、8 エントリのノンブロッキング・キューを 1 つ備えている。L3 キャッシュはオンダイに統合されており、最大帯域幅は 32GB/ 秒である。L2（およびシステム・バス）への転送には 4 サイクルを要する。このため、レイテンシは最小で 12 サイクルで、キューが埋まるとさらに増える。L3 キャッシュにヒットしないデータ・アクセスはシステム・バスとチップセットを経由してメイン・メモリにアクセスする。

メモリ・アクセスと L2 OzQ

L1 データ・キャッシュと L2 キャッシュはそれぞれ専用のマイクロパイプラインを経由してコア・パイプラインおよびレジスタ・ファイルと接続されている。ストール、キャンセル、リダイレクトなどが発生すると、レジスタ・ファイルへのスムーズなデータ・フローが妨げられ、機能ユニットの利用効率が低下し、コア・パイプラインにストール・サイクルが発生する。なお、L2 でアウトオブオーダーのデータ・アクセスが可能になっているのは、L2 リクエスト・キューの働きによるものである。ここでは、この L2 キューについて解説を行う。

Itanium® 2 プロセッサのキャッシュはすべてノンブロッキング・キャッシュで、データをアウトオブオーダーに返せる。これは、L1 データ・キャッシュのみがノンブロッキングであった Itanium プロセッサからの変更点である。Itanium 2 プロセッサでは、L2 キャッシュへのアクセスはアウトオブオーダー・キュー（OzQ）を使ってスケジューリングされる。OzQ のエントリ数は 32 である。OzQ は L1-D で処理できなかった要求を処理する。これには、すべてのストア、セマフォ、浮動小数点データのロード、キャッシュ不可なアクセス、L1-D ロード・ミス、L1-D で未解決の競合が含まれる。L2 キャッシュの設計上、競合さえなければ OzQ エントリの数が 32 個もあれば、L1D 要求を十分に保持できるはずである。しかし実際には L2 内では多くの競合が発生する。このような競合（バンク競合、1 つのキャッシュ・ラインに対する複数のキャッシュ・ミスなど）が発生すると、要求が OzQ 内部にとどまる時間が長くなる。OzQ のエントリ数を増やせば、L2 キャッシュが競合を解決している間に、L1-D パイプライン（および FP ロード）はヒットの処理を続け、L2 に対する次の要求を送り出せる。競合があると L2 のレイテンシが大きくなり、複雑なアプリケーションにおける L2 レイテンシの予測が不可能になる。

競合やリソースが利用できないことが原因で L2 OzQ がフルになるときもある。このような場合、L1-D パイプラインは OzQ に入っている要求のいずれかが満たされ、キューに空きができるまでストールする。これによってコア・パイプラインもストールする。

OzQ の制御ロジックは、直前のサイクルで割り当てられた最後のエントリ（テール）から始めて、1 サイクルごとに最大 4 つの連続するエントリを割り当てる。使用可能なエントリの数が少ない場合（4 から 12 の間）は、操作がそれ以上 L2 に渡されないように、L1-D パイプラインがストールする。OzQ に入っている要求の処理が L2 内で完了すると、要求は OzQ から削除される。すなわち、以下の場合である。

- ストアがデータ配列（L2 キャッシュ）を更新したとき
- ロードが適切なデータをコアに返したとき
- L2 ミスの要求がシステム /L3 によって受け入れられたとき

OzQ 制御ロジックにはラウンドロビン方式のポインタが3つ用意されており、それぞれがヘッド、テール、イシューの位置を追跡する。新しいアクセス要求が OzQ に与えられると、最後の OzQ エントリの場所を示すテール・ポインタの位置に割り当てられる。ヘッド・ポインタは OzQ 内で最も古いエントリの場所を示す。イシュー・ポインタは常にヘッドとテールの両ポインタの間にあり、イシュー・ロジックが次に L2 パイプラインに送るエントリの場所を示している。このようにヘッド / テール方式で管理を行っていると、ヘッドとテールの間にある命令が発行済みとなると無効なエントリが「穴」のように発生する。OzQ では、こうした穴を埋めるための圧縮を行わない。穴の部分には新しい L1-D や FP ロード / ストア要求を格納することができないため、有効な OzQ エントリはわずかしか存在しないにもかかわらず、OzQ 制御ロジックがフルになって新しい L1-D や FP 要求を受け付けなくなることがある。

このように無効なエントリの穴が開くなどして OzQ がフルになると、OzQ に十分な空きが生まれるまでパイプラインはストールする。このような場合は、競合（キャッシュ・ミスした同一のラインに対する複数のアクセスや、バンク競合など）を引き起こす可能性のあるデータ要求の順番を変更して競合を回避し、OzQ の再循環やキャンセルを削減する必要がある。通常、これはソース行の順序を入れ替えたり、明示的にループ・アンローリングを行うだけで解決する。

また、キャッシュ・ミスが起こると、エントリが OzQ 内部にとどまる時間が長くなる。プリフェッチ組み込み関数を使えば、キャッシュ・ミスを減らし、OzQ がフルになって発生するストール・サイクルを低減できる。

L2/L3 がアウトオブオーダーにデータを返すことにはもう1つ副作用がある。それは、Itanium プロセッサのブロッキング L2 キャッシュに依存した同期化コードを作成していると、Itanium 2 プロセッサでは同様の同期化が行われないため、コードの実行時エラーを引き起こす可能性がある点である。この問題は、コーディングの方法に原因がある。EPIC アーキテクチャでは、メモリ・フェンスに関するセマンティックスに正しく従わなければならない。これについては、『インテル® Itanium® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル第 2 巻』第 2 章を参照のこと。

仮想アドレスから物理アドレスへの変換

プロセッサが物理メモリにアクセスするには、仮想アドレスから物理アドレスへの変換が必要である。この変換は、2 階層のデータ・トランスレーション・ルックアサイド・バッファ（DTLB）とハードウェア・ページ・ウォーカ（HPW）で構成される階層システムで行っている。データ・アクセス時にこれらテーブルを更新したりページ・ウォーカを呼び出したりすると、データ・アクセス時のレイテンシはきわめて大きくなる。

トランスレーション・ルックアサイド・バッファと ハードウェア・ページ・ウォーカ

2 階層のデータ・トランスレーション・ルックアサイド・バッファ（DTLB）は、仮想アドレスから物理アドレスへの変換を行うハードウェア・チェーンの一部である。これらの DTLB は、ハードウェア・ページ・ウォーカがアクセスする仮想ハッシュ・ページ・テーブル（VHPT）に対する 2 階層のキャッシュ構造として、物理アドレスへの変換を行う。仮想メモリ・システムの詳細は、『インテル® Itanium® 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 第 2 巻』を参照のこと。

目的のデータが L1 データ・キャッシュに格納されていたとしても、1 次のデータ TLB（L1 DTLB）内に該当のエントリがなければ、L1 キャッシュにはヒットしない。このように、L1 DTLB ミスは L1 データ・キャッシュ・ミスを引き起こすため、TLB ミスはストール・サイクルの原因となる。L1 DTLB は L1 データ・キャッシュと同時にアクセスされるため、L1 データ・キャッシュから整数データをロードする際のレイテンシは前述のとおり 1 サイクルである。また、ストアおよび浮動小数点ロードが L1 DTLB ミスになっても、ペナルティは発生しない。

L1 DTLB は 32 エントリ、フル・アソシアティブ、デュアルポート・バッファであり、4KB ページのみをサポートする。これよりも大きなページについては、複数のエントリで対応する。L1 DTLB のページ変換データが置き換えられると、それに対応する L1 キャッシュ内のエントリはすべて無効になる。

2 次の DTLB（L2 DLTB）は 128 エントリ、フル・アソシアティブ、4 ポート・バッファである。L2 DTLB は L1 DTLD と並行してアクセスされ、仮想アドレスから物理アドレスへのマッピングを行うとともに、保護情報も提供する。整数ストアと浮動小数点ロードが L1 DTLB ミスになっても、ペナルティは発生しない。整数ロードが L1 DTLB にミスして L2 DLTB にヒットした場合は、L2 DLTB からデータを転送するために 4 サイクルのペナルティが発生し、キャッシュ・ラインを更新するためにさらに 1 サイクルのペナルティが追加される。

仮想アドレスから物理アドレスへの変換を行う階層構造の 3 番目に位置するのが、仮想ハッシュ・ページ・テーブル（VHPT）で、このテーブルにはハードウェア・ページ・ウォーカ（HPW）がアクセスする。L2 DLTB ミスが発生すると、HPW が VHPT にアクセスして仮想アドレスから物理アドレスへの変換データを見つけ、TLB を更新する。この後、データ・アクセスは通常どおり完了する。

VHPT は OS によって管理されており、キャッシュ可能なメモリ内に置ける（OS によって異なる）。VHPT データ・ページの位置は必要に応じて HPW に知らせる必要がある。これらの位置も DTLB システム内に「キャッシュ」できるため、高速なアクセスが可能である。VHPT データの位置が L2 DTLB 内になければ、VHPT フォルトが生成され、OS が HPW に対して変換データを提供する。

多くの OS では VHPT ショート形式を使用しているが、その場合、VHPT は 1 エントリにつき 8 バイトを使用するため、8196KB ページの VHPT データごとに 1024 個の VHPT エントリが存在することになる。VHPT ページを検索する際、HPW は VHPT データにアクセスして、VHPT エントリの 1 ページに対して L2DTLB エントリを 1 つ使用する。VHPT データがキャッシュ可能であれば、OS がメイン・メモリからデータ構造にアクセスしなければならない場合に比べ、はるかに高速に VPW が DTLB システムを更新する。

L2DTLB には 128 エントリがあるが、これらすべてをアプリケーションのアドレス可能な領域へのアクセスに使用できるわけではない。OS はこれらエントリの最大半分までを変換レジスタとして確保できる。しかし実際に確保される数はそれほど多くない。また、VHPT の検索のためにエントリの一部を使用する場合もある。このため、遠くのデータ・ページをどれだけ同時にアクセスしているかによって、L2DTLB で利用可能なエントリの数は最大値の 128 よりも大幅に少なくなるときがある。

必要な変換データが TLB 内に存在しない場合は、VHPT フォルトが生成され、OS ハンドラが VHPT から変換データを探すことになる。通常、このハンドラはきわめて高速に動作するが、それでも完全にハードウェアで行われるアクセスに比べれば速度は大きく低下する。

VTune™ パフォーマンス・アナライザによるイベント・ベースのサンプリング

4

VTune™ パフォーマンス・アナライザのイベント・ベースのサンプリング（EBS）機能は、あるコンピュータ上でソフトウェアを実行させたときに発生するマイクロアーキテクチャ・イベントをベースとしたサンプリングを行う。これらイベントは、プロセッサのハードウェア・パフォーマンス・カウンタを使って測定される。EBS で生成されるデータを利用すれば、ソフトウェア実行時のマイクロプロセッサの動作を詳細に知ることができる。EBS はマイクロアーキテクチャ・イベントがある一定回数発生するたびにプロセッサの実行コンテキストをサンプリングする。マイクロアーキテクチャ・イベントの例としては、分岐予測ミスやキャッシュ・ミスなどがある。EBS のデータ収集が完了すると、VTune アナライザはプロセス、スレッド、モジュール、そしてコードがデバッグ・シンボルを生成するようにコンパイルされている場合は関数、コード行ごとにイベントを表示する。

実行コンテキストのサンプリングはイベントがある一定回数発生したら行われるが、このときのイベント発生回数を Sample After 値と呼ぶ。ここで注意が必要なのは、イベントが 1 回発生するたびに（すなわち、パフォーマンス・カウンタの値が 1 つ増えるたびに）サンプルの収集が行われるのではない点である。VTune アナライザでは Sample After 値も自動的に設定できるが、GUI を使ってユーザが手動でも明示的かつ強制的に値を設定できる。VTune アナライザが Sample After 値を自動で決定する場合は、その値を計算するためにデータ収集を 1 回余分に行う必要がある。このプロセスをキャリブレーションと呼ぶ。VTune アナライザは、平均して 1 ミリ秒ごとに 1 サンプルを収集するように Sample After 値のキャリブレーションを行う。

VTune アナライザはサンプリング・データを収集したらその結果を棒グラフまたは表形式で表示する。サンプルは、プロセス、スレッド、モジュール、関数、ソース行単位で表示が可能である。ただし、関数およびソース行単位でサンプルを表示するには、デバッグ情報がなければならない。デバッグ情報がない場合は、ユーザがドリルダウンできるのはアセンブリ・コードのみとなる。

Itanium® 2 プロセッサの場合、サンプリングされた命令プログラム・カウンタには、動的命令数にして約 48 個分ほどの誤差が認められるときがある。つまり、Itanium 2 プロセッサの場合、実際にはソース・ビューに表示されているよりも前のアドレスでイベントが発生していることになる。これを「イベント・スキッド」と呼ぶ。EAR（Event Address Register）以外のイベントはすべてイベント・スキッドの影響を受ける。このため、EAR 以外のイベントを使用する場合は、イベントの発生個所をコード行単位ではなくコード・ブロック単位でとらえる必要がある。例えばループの場合は、ループ内の特定のコード行やアセンブリ単位ではなく、ループ全体でイベント・データを観察する必要がある。

VTune アナライザでは、プロジェクトとアクティビティの階層を使って分析結果を管理している。プロジェクトとは、複数のアクティビティをまとめたものである。また、1 つのアクティビティにはデータ・コレクタの設定や分析結果をいくつも含められる。同一プロジェクト内に複数の結果がある場合、プロジェクト・ナビゲータ・ウィンドウに表示されている 1 つの分析結果を、別の分析結果のサンプリング・データ・ビューへドラッグ・アンド・ドロップすれば、これらの結果どうしを比較できる。

このほか、VTune アナライザには Windows* および Linux* に対応したリモート・サンプリング・データ収集（RDC）機能も用意されている。この機能を利用すると、ユーザは VTune アナライザの GUI によるオーバーヘッドのない環境でサンプリング・データを収集できる。Windows の場合は、RDC は DCOM を使用する。Linux の場合は、TCI/IP 上で独自のプロトコルを使ってリモート・サンプリング・データを収集する。

なお、1 台のコンピュータ上で同時に複数のインスタンスの VTune アナライザを実行して、データのサンプリングは行えない。また、RDC にはリクエスト・キューイング機能が用意されていないため、1 度にシステムのプロファイリングを実行できるのは 1 ユーザに限られる。

EBS をデフォルトのまま実行すると CPU_CYCLE（Clockticks とも呼ぶ）および Instructions Retired（リタイアした命令数）のデータが収集され、1 命令をリタイアするのに必要なサイクルが算出される。イベント名の言及には、常にショート形式を使用する。これについては本書でも簡単に解説するが、詳しくは『インテル® Itanium® 2 プロセッサ・リファレンス・マニュアル：ソフトウェアの開発と最適化』を参照のこと。また、umask の値を設定するとアクセスできるサブコンポーネント（詳しくは第 5 章を参照）についても、あらかじめすべて定義されている。

サンプリングを行うイベントをデフォルト（Clockticks および Instructions Retired）から変更するには、EBS ウィザードでサンプリング・コレクタの設定を行う。あるいは、1 回サンプリングを実行した後で、プロジェクト・ナビゲータ・ウィンドウに表示されているアクティビティを右クリックして、メニューから [Modify Activity] を選択してもよい。一般的には、アクティビティのコピーを作成してから、そのコピーに対して変更を加えるのが好ましい。そうすれば、これまでに収集したデータをプロジェクト内で引き続き利用できる。新しいイベントを選択したら、[Start Activity] ボタン（緑の右矢印）をクリックすると、キャリブレーションが行われ、その後で新しいアクティビティのサンプリングが開始される。詳しくは、VTune アナライザのオンライン・ヘルプ『Working with Projects and Activities』を参照のこと。

制限事項

- 1 回のサンプリング実行で最大 4 つのイベントに関するデータを収集できる。このため、キャリブレーションとデータ収集を行うとアプリケーションの実行回数がきわめて多くなる。
- 現在のところ、VTune™ アナライザのインターフェイスは 32 ビット・アプリケーションである。このため、多くのイベントに関するデータを収集しようとすると、OS がデータ収集またはキャリブレーションを 1 回実行するたびにプロセッサのステートを 32 ビット・モードと 64 ビット・モードの間で切り替えるため、きわめて高い負荷が発生する。このように、いくつものカウンタに関するデータを収集しようとすると、サンプリング実行回数が増え、十分な精度の結果が得られない場合がある。データ収集プロセスの負荷が少ないリモート収集モードの利用を推奨する。

パフォーマンス監視とサイクル・アカウンティング 5

マイクロアーキテクチャ最適化の目的は、CPU の機能ユニットへの命令フローを最大化することにある。これは、コア・パイプラインのストール・サイクルを最小化することに等しい。これを体系的に実現するには、アプリケーションの実行時に発生するパイプライン・ストールの原因を定量的に分類していく必要がある。これにより、パイプライン・ストールの最大の原因を効率的に改善できる。

Itanium® プロセッサでは、パイプラインをストールさせる可能性のあるステージにそれぞれ 1 つのイベントが関連づけられており、これによってそのステージで発生するストール・サイクルをカウントできるようになっている。また、これらのイベントには優先度が設定されており、複数のパイプライン・ステージで同時にストールが発生した場合は、その中で最も優先度の高いものに対してストール・サイクルが割り当てられる。ストールの優先度はパイプラインの終わりに近づくにつれて高くなる。このような仕組みにより、いくつかのアーキテクチャ・サブシステムで発生したストール・サイクルをすべて合計すると、ストールした CPU サイクル全体に等しくなる総和則が成り立つ。サイクル・アカウンティングの各内訳はそれぞれ異なるアーキテクチャ機能に関連づけられているため、どのアーキテクチャ機能が実行効率の低下を招いているかを容易に把握できる。

Itanium プロセッサでは、機能ユニットに命令を発行する際に費やしたサイクルもイベントを発生し、明示的にカウントされる。これとは異なり、Itanium 2 プロセッサでは機能ユニットに対して命令を発行するのに費やしたサイクルは他のイベントから計算しなければならない。

マイクロアーキテクチャへの最適化によるパフォーマンス・チューニングとは、サイクル・アカウンティングの内訳のうち、パイプライン・ストールに関係したものを最小化していく作業となる。つまり、命令を機能ユニットに発行するのに費やさなかったサイクルを最小化する。

Itanium® 2 プロセッサにおけるサイクル・アカウンティングの総和則

BACK_END_Bubble.ALL イベントは、命令パイプラインが何らかの理由によってストールしたサイクルをすべてカウントする。したがって、命令を機能ユニットに発行するのに費やしたサイクル数は以下で求められる。

```
CPU_Cycles - Back_End_Bubble.ALL
```

また、パイプラインのストールの内訳を分類すると、次の総和則が成り立つ。

```
Back_End_Bubble.ALL =
 BE_Flush_Bubble
+ BE_L1D_FPU_Bubble
+ BE_EXE_Bubble
+ BE_RSE_Bubble
+ Back_End_Bubble.FE
```

上記のコンポーネントは優先度の高いものから順に（すなわち、コア・パイプラインの下流から上流に向かって）記載している。各コンポーネントはそれぞれ異なるアーキテクチャ・サブシステムによって発生するストール・サイクルをカウントする。Itanium® プロセッサのパフォーマンス監視イベントも、このように優先度をつけてカウントを行う点は同じである。表 5-1 は、ストール・サイクルをカウントするためのイベントを Itanium 2 プロセッサと Itanium プロセッサで比較したものである。

表 5-1　Itanium® 2 と Itanium® プロセッサにおけるサイクル・アカウンティング用イベントの比較

Itanium® 2 プロセッサのサイクル・アカウンティング用イベント	Itanium® プロセッサのサイクル・アカウンティング用イベント
BE_Flush_Bubble	Pipeline_Backend_Flush_Cycle
BE_L1D_FPU_Bubble BE_EXE_Bubble	Data_Access_Cycle Dependency_Scoreboard_Cycle
BE_RSE_Bubble	RSE_Active_Cycle.d
Back_End_Bubble.FE	Unstalled_Backend_Cycle Inst_Access_Cycle Taken_Branch_Cycle.d

Itanium プロセッサおよび Itanium 2 プロセッサのサイクル・アカウンティング用イベントを公平に比較するには、これらのイベントをパイプライン・ストールの原因が同じものどうしでグループ化する必要がある。例えば、Itanium 2 プロセッサの BE_L1D_FPU_Bubble イベントと BE_EXE_Bubble イベント、Itanium プロセッサの Data_Access_Cycle イベントと Dependency_Scoreboard_Cycle イベントなどである。どちらの場合にも、これら 2 つのイベントを 1 つのセットとして考えれば、メモリ・アクセスによるストールとスコアボード・レジスタ依存性によるストールを調べられる。Itanium プロセッサの場合は、メモリ・アクセスによるストールとスコアボード依存性によるストールはそれぞれ Data_Access_Cycle と Dependency_Scoreboard_Cycle の専用のカウンタで計測できる。一方、Itanium 2 プロセッサではアーキテクチャ・サブシステムに密接に関係したカウンタでこれらのストールを計測する。

Itanium プロセッサの RSE_Active_Cycle.d カウンタの値は、次式を用いれば他のカウンタから求められる。

```
Memory_Cycle - Data_Access_Cycle
```

本書の目的は、各カウンタを紹介し、アプリケーション最適化でこれらカウンタを体系的に使用する方法について説明することである。サイクル・アカウンティング・アプローチの長所は、アーキテクチャの使用効率の低下を招いている原因を直感的な方法で相対評価できる点にある。まず上記の 5 つのコンポーネントでストールの要因を全体的に把握し、次に最も大きな要因となっているコンポーネントをさらに詳細に分析し、命令の実行以外に費やされている CPU サイクルの原因を特定していく。

サイクル・アカウンティングの総和則を構成する 5 つのコンポーネント（イベント）はそれぞれ複数のサブコンポーネント（サブイベント）で構成されており、これらのサブコンポーネントには umask（ユーザ・マスク）の値を設定すればアクセスできる。これについては別の技術ドキュメント[1] で詳しく解説されているが、イベント名に「コンプリータ」と呼ばれるフィールドを追加して識別される。例えば、Back_End_Bubble は親イベントである。これは、umask = 0 をデフォルトとして暗黙的に指定している。Back_End_Bubble.all は親イベントと同じ意味を持つ。「.all」のコンプリータを付けると umask = 0 を明示的に指定している点のみが異なる。Back_end_Bubble.FE イベントは、「.FE」のコンプリータを付ければ、umask = 1 を明示的に指定したサブイベントである。一般に、サイクル・アカウンティングの監視イベントのサブコンポーネントは、親イベントのような優先づけはされてなく、また、サブイベントを合計しても親カウンタ（通常、umask = 0）と厳密に一致するとも限らない。サブイベントの詳細は、サイクル・アカウンティングの総和則を構成する個々のコンポーネントを解説した章で詳しく取り上げる。

Itanium プロセッサのパフォーマンス・イベントについては、VTune™ アナライザのインターフェイスで umask の値を設定できる（[Configure] メニューでイベントを選択し、選択したイベントが反転表示されている状態で [Edit event] ボタンをクリックする）。ただし、Itanium 2 プロセッサのパフォーマンス・イベントの場合は、サブイベントがすべて明示的にあらかじめ定義されているため、umask を選択する必要はない。

1. 『インテル® Itanium® 2 プロセッサ・リファレンス・マニュアル ソフトウェアの開発と最適化』参照。

アプリケーションの分析

サイクル・アカウンティングを行うと、マイクロアーキテクチャ内におけるアプリケーションのフローを体系的に分析できる。サイクル・アカウンティングの総和則を利用すれば、マイクロプロセッサのパイプラインではどのアーキテクチャ・サブコンポーネントが命令のフローを妨げているのかを一目で理解できる。この機能を使って、重要な実行ボトルネックの発生源を評価できる。分析を行う際は、サイクル・アカウンティングの総和則を構成するコンポーネントの優先度の高いものから順に行っていく。以下、本書では各コンポーネントについて解説を行うとともに、ストール状態を発生させる最も一般的な原因についても説明を加えていく。

アプリケーションの最適化で最も重要なのは、パフォーマンス監視イベントを使ってストールの最大の原因を特定し、その相対的な重要性の判断である。キャッシュ・ミスを減らしても、それがパフォーマンス低下の原因としてほとんど関係していなければ、それは非効率的で無駄な作業に終わってしまう。したがって、アプリケーション実行のストールに対する影響の大きさに着目してパフォーマンス上のボトルネックを優先づけし、重要性の高いものから順に解決していくことが肝要である。

なお、上述の総和則はそれだけで十分な情報を得られるが、正規化した比率として各コンポーネントを表した方が、その影響を理解しやすい。ただしここでCPU_CYCLES に対して正規化を行うのは浅慮である。これではゼロ・サム・ゲームになってしまい、分析作業がかえってわかりにくいものとなってしまう。一般的には、処理した作業量（アプリケーションに依存した定義）、あるいはリタイアした Itanium® 命令（イベント・コード = 8）に対して正規化を行うのがよい。本書では、カウントされたデータを IA64_Inst_Retired（IA64IR）イベントに対して正規化する。これは IA64_Tagged_Instructions_Retired のサブイベントであり、プレディケートが真であるすべての命令、プレディケートの値にかかわらずすべての分岐命令をカウントする。これは、Itanium プロセッサから若干変更になっている点である。

アプリケーションの実行効率を分析する際に CPI（Cycles Per Instruction）を使用すると、簡単な代数計算を行うだけでストールに関するサイクル・アカウンティングの各コンポーネントにドリルダウンできる。IA64_Instructions_Retired（IA64IR）の値はすべてのコンポーネントに共通しており、しかも最適化作業を進めていっても比較的変動が少ないため、この値を分母にすると簡単に分析を行える。

```
CPI = CPU_CYCLES/IA64IR
```

アプリケーションのアルゴリズムを変更したり、コンパイラのフラグを使用して最適化を行うと、命令の総数（経路の長さ）が変化し、CPI の値も変化する。それでもなお、実行効率を標準化して比較可能な形にするには、CPI に対して正規化するのがよい。

マイクロアーキテクチャ最適化の目的は、ストール・サイクルの削減である。ストールを削減できたかどうかは、Back_End_Bubble イベントを使って測定できる。CYC_RET_INST を命令のリタイアに費やしたサイクル数と定義すると、次式が成り立つ。

```
CYC_RET_INST = CPU_CYCLES - Back_End_Bubble
```

これは Itanium プロセッサの All_Stops_Dispersed イベントに等しい。つまり、次式が成り立つ。

```
CPI = CYC_RET_INST/IA64IR + Back_End_Bubble/IA64IR
```

および

```
Back_End_Bubble/IA64IR = ΣBE_Bubble_COMPONENTS/IA64IR
```

この BE_Bubble_Components（サイクル・アカウンティングの総和則を構成する各コンポーネント）は、サブイベントを用いるか、あるいは他のアーキテクチャ監視イベントの発生（「発生イベント」と呼ばれる。詳細は後述）に対するペナルティを組み込んだモデルでさらに細かく分類できる。これは特に、メモリ・アクセスによるストールをカウントする BE_EXE_Bubble と BE_L1D_FPU_Bubble に当てはまる。つまり、以下の関係式が成り立つ。

```
BE_Bubble_Component/IA64IR ~
ΣOccurrence_Events*Penalty/IA64IR
```

このように、アーキテクチャ監視イベント（あるいはこれらイベントのカウント数の差）の合計は、対応する BE_Bubble ストール・カウンタで計測されたパイプライン・ストールに対する影響の大きさと関係がある。

この最後のステップは、Itanium プロセッサの場合ほど正確ではない。これは、L2 キャッシュがアウトオブオーダーにデータを返し、OzQ 内で複雑なスケジューリングが行われるためである。Itanium プロセッサの L2 FIFO キューの場合は、上記の方法できわめて正確なモデリングが可能である。ところが Itanium 2 プロセッサの場合は OzQ からデータがアウトオブオーダーに返ってくるため、アクセス・ペナルティが互いに隠蔽しあう。このため、メモリ・アクセス・モデリングが複雑化する結果となっている。それでもなお、メモリ・アクセス・モデリングはどの部分を改善すれば最も大きな効果が得られるかを判断する上で有効な手段であるのには変わりはない。

サイクル・アカウンティングのコンポーネント

この節では、サイクル・アカウンティングの総和則を構成する各コンポーネントの概要を解説する。これらコンポーネントの詳細は、第 6 章から第 10 章で詳しく解説する。

BE_Flush_Bubble

このカウンタは、DET ステージで発生したストール・サイクルをカウントする。プロセッサ・パイプラインのステージのうち、WRB ステージではストールは発生しないので、パイプラインの最も下流でストールを発生する可能性があるのがこの DET ステージである。ストール発生の原因となるのは次の 2 つで、それぞれ優先度が設定されている。

- 例外によるパイプライン・フラッシュ（be_flush_bubble.xpn：umask の値は bxx10）
- 分岐予測ミスによるパイプライン・フラッシュ（be_flush_bubble.bru：umask の値は bxx01）

例外や分岐予測ミスによるパイプライン・フラッシュは、フロントエンド・パイプラインのストールに関連づけられる場合もある。コンパイラはコード・フローを予測しながらコードを生成しており、これが分岐予測ミスや例外を招くことになる。また、コンパイラは予想されるコード・ブロックどうしをバイナリ内でまとめて配置し、命令キャッシュのアクセスおよび格納効率を上げようとする。このため、パイプライン・フラッシュが発生すると、あまり頻繁に使われないコード・ブロックを探すことになるため、命令キャッシュ・ミスが起こり、パイプラインのフロントエンドがストールする。このカウンタについては、第 8 章で詳しく解説する。

BE_L1D_FPU_Bubble

BE_L1D_FPU_Bubble カウンタは、L1-D および FPU マイクロパイプラインのストールによってコア・パイプラインの DET ステージに発生したストール・サイクルをカウントする。実際には、FPU マイクロパイプラインが原因でストール・サイクルが発生することはあまりない。L1-D マイクロパイプラインに起因するストールは、データ・フローの渋滞、DTLB ミス、HPW の動作によって発生する。このため、メモリ・アクセスによるストールも一部このカウンタで記録される。BE_L1D_FPU_Bubble カウンタの詳細については、第 7 章で解説する。

BE_EXE_Bubble

BE_EXE_Bubble カウンタは、パイプラインの EXE ステージにおけるストール・サイクルをカウントする。これには、メモリ・アクセス・レイテンシによるストールやスコアボード依存性によるストールが含まれる。メモリ・アクセスによるストールの大部分はこのカウンタで記録される。BE_EXE_Bubble カウンタの詳細については、第 6 章で解説する。

BE_RSE_Bubble

BE_RSE_Bubble カウンタは REN ステージにおけるストール・サイクルをカウントする。REN ステージのストールは、利用可能なリソースが十分にない場合に発生する。この場合、レジスタ・スタック・エンジン（RSE）がバッキング・ストア・メカニズムを起動して物理レジスタを解放し、汎用レジスタ・スタックに空きを作る。コア・パイプラインの REG ステージは、必要な RSE ロード / ストアを送り込んでレジスタ・スタック・エンジンを起動する。一般に、プロファイルに基づくフィードバック（Qprof_gen/Qprof_use）およびプロシージャ間のインライン化（Qipo）オプションを使用して再コンパイルすれば、レジスタの利用効率は大幅に改善される。このほか、きわめて頻繁に使用される関数チェーンの引数の数を構造体の使用によって減らしたり、再帰的アルゴリズムへの依存性を少なくすることも、レジスタ割り当てに対する負担を軽減する上で効果的である。BE_RSE_Bubble カウンタの詳細については、第 9 章で解説する。

Back_End_Bubble.FE

Back_End_Bubble.FE カウンタは、ディスパーサルすべき命令がないためにバックエンド・パイプラインがストールした場合のサイクルをカウントする。パイプラインのフロントエンドが要求された時間内に命令を供給できない場合は、コア・パイプラインのバックエンドにストール・サイクルが発生する。このストールは最も優先度が低いため、Back_End_Bubble.FE カウンタがカウントされたのは、それ以外のストールは一切発生していないことを意味する。Back_End_Bubble.FE でカウントされるストールの典型的な原因としては、コンパイラがアクティブな命令ブロックを適切にグループ化しておらず、L1-I キャッシュの使用効率が低いと考えられる。もちろん、前述のとおり分岐予測ミスの結果としても発生するが、動的分岐予測ハードウェアが正しく分岐予測を行っていても発生する場合がある。通常、プロファイルに基づくフィードバック（Qprof_gen/Qprof_use）やプロシージャ間のインライン化（Qipo）で再コンパイルを行えば、この種のストールには大きな効果が期待できる。Back_End_Bubble.FE カウンタの詳細については、第 10 章で解説する。

コア・パイプラインのストールの内訳

サイクル・アカウンティングと発生イベント

Itanium® プロセッサ・ファミリのサイクル・アカウンティング機能の基本は、コア・パイプラインのストール・サイクルをカウントすることにある。1 つのサイクルで複数のストール状態が発生した場合は、パイプラインの最も下流の（最も優先度が高い）ステージのストールとしてカウントされる。さらに、その他のマイクロアーキテクチャ・イベント・カウンタ（発生イベント）を使ってストール・サイクルの原因をより詳細に調べられる。発生イベントとは、特定のアーキテクチャ機能の動作をカウントするものである。

ここで、整数データのロードが L1 データ・キャッシュにミスし、L2 キャッシュにヒットした場合を考えてみる。このとき、アーキテクチャ・イベントとしては L1 データ・キャッシュ・ミスが 1 回発生し、発生イベント L1D_READ_MISSES のカウンタの値が 1 つ加算される。ところが、キャッシュ・ミスによってロード時のレイテンシも大きくなるため、命令がこのデータに依存している場合は、データが利用可能になるまで EXE ステージがストールする可能性がある。そして、データの読み出しが完了し、機能ユニットがこのデータを利用できるようになるまでのストール・サイクルを BE_EXE_Bubble カウンタが計測したとする。すると、このとき例外や分岐予測ミス、DTLB ミスなどによってパイプライン・フラッシュやマイクロパイプラインのストールは発生していないことがわかる。これらの原因によるストールは、パイプラインのより下流に位置する DET ステージで発生するため、仮にこれらのストールが発生していれば、BE_EXE_Bubble よりも優先度の高いカウンタ（BE_FLUSH_Bubble または BE_L1D_FPU_Bubble）がストール・サイクルを計測するはずである。

このように、パフォーマンス・モニタでサイクル・アカウンティングを行う場合、ストール・サイクルは優先度の最も高いコンポーネント 1 つだけに割り当てられる。一方、発生イベント・カウンタについては、1 サイクルでいくつものカウンタの値が同時に増えることがある。このため、発生イベントのデータを分析するのは複雑な作業となり、慎重に行う必要がある。ここでもう一度上の例を考えてみる。EPIC アーキテクチャでは L1 データ・キャッシュにミスしている間もアウトオブオーダーにメモリからデータが返ってくるときがある。このため、メイン・メモリにアクセスする必要のある別のロード命令が同時に発生しており、同じサイクルで L1-D、L2、L3 でキャッシュ・ミスが起こる可能性も考えられる。

発生イベント

これらの発生イベント・カウンタについては、それぞれの発生イベントが影響を及ぼす、優先度の設定されたストール・サイクル・カウンタの下位に属するものとして取り扱われる。このように、発生イベントがストール・サイクルのカウント数にどのように影響しているかを調べられる。さらに重要な点として、このように体系づけることによって、EBS による分析プロセスを忠実に反映したイベントのグループ化が可能になる。

個々の発生イベントによって生じたストール・サイクルのペナルティを調べるには、テスト・プロシージャをきわめて慎重に作成する必要がある。一般に、テスト・ケースはアセンブラで作成する。コンパイラはレイテンシを隠蔽してパイプラインを効果的に利用しようとするためである。

発生イベントの説明の中で紹介するテスト・ケースを見れば、どのようなコーディング構造がストールを引き起こすのかがわかる。また、ストールのペナルティの測定方法もテスト・ケースの中で明示的に示される。

サブイベント

ほとんどすべてのパフォーマンス監視カウンタは、サブイベントを利用することによって、より詳細なサブコンポーネントに細分される。各イベントには一意のイベント・コードがあり、サブコンポーネントは（0 以外の）ユーザ・マスク（umask）を設定すれば指定される。umask に 0 を設定すると、通常はすべてのサブイベントに対するデータを同時に収集する。1 サイクルでいくつものサブイベントが発生しても、親イベント（umask=0）のカウント数は 1 つしか増えない。したがって、サブイベントのカウント数を合計しても必ずしも親イベントのカウント数と一致するわけではない。しかし一般的には、サブイベントのカウント数を合計したものが親カウンタの値と等しくなる近似を行っても、プログラムの実行効率を理解する上で十分な精度の情報が得られる。これは、サイクル・アカウンティング・イベントにも当てはまる。各サブコンポーネント（umask != 0）の合計は親カウンタ（umask=0）の値と等しくなることは保証されていないが、実際には多くのイベントで、サブイベントの合計が親カウンタの値と等しくなる。例えば、L2_Data_References.L2_All は親カウンタである（ただし umask=3）が、この値は L2_Data_References.L2_Data_Reads（umask=0x01）と L2_Data_References.L2_Data_Writes（umask=0x02）の合計に一致する。

分析は主に VTune™ パフォーマンス・アナライザを使って行われるが、VTune アナライザは統計的サンプリングを行うため、正確性と不正確性は通常統計的誤差の範囲内に収まる。デフォルトの umask の値は、VTune アナライザの GUI で変更できる（イベント・ウィンドウで目的のイベントを選択し、[Edit event] ボタンをクリックする）。VTune アナライザ用の Itanium® 2 プロセッサ・パックでは、ユーザの利便性を高めるためにすべてのサブイベントはあらかじめ特定の umask でプログラミングされているため、この手順は必要ない。

イベント・スキッドと EAR イベント

データ収集モード中、特定のイベントが発生して割り込みが起こると、VTune アナライザはそのときの命令ポインタ（IP）の位置を記録するが、このとき IP の記録精度には幾分の誤差が生じる。このように、プログラム内における本来のイベント発生位置とは若干異なる位置として記録してしまうことをイベント・スキッドと呼ぶ。通常はイベント・スキッドが大きな問題になることはないが、IP の位置に関して高い精度が求められる場合のために、EAR（Exact Address Register）イベントと呼ばれるパフォーマンス・イベントが用意されている。これは、イベント発生時にハードウェアが IP を記録するものである。EAR イベントには 6 種類あり、L1 命令と ITLB ミス、L1 データと DTLB ミス、ALAT ミス、フロントエンド・ストールに関するデータを収集できる。

マイクロベンチマークによるアーキテクチャの研究

マイクロベンチマークとは、プロセッサの動作を意図的に制御する「微細な」テスト・ケースを作成するために使用するサンプル・プログラムである。個々のアーキテクチャ・イベントを繰り返しループの中で発生させることによって、特定の発生イベントに対するストール・サイクルのペナルティを正確に測定できる。また、ある特定の状況でどのイベントやサブイベントが発生するかを、意図的に制御しながら調べられるのもマイクロベンチマークの利点である。こうして得た知識は、実際のアプリケーション分析にも応用できる。ただし、実際のアプリケーションとマイクロベンチマークでは複雑なアーキテクチャやキューイングの利用状況も大きく異なるため、マイクロベンチマークの結果をそのまま当てはめられない部分も大きい。しかし、まずマイクロベンチマークから分析を始めていくのは妥当な方法であるし、イベント・ベースのサンプリングによるマイクロアーキテクチャ最適化手法の基礎を学ぶにはマイクロベンチマークは十分な役割を果たす。

実行効率低下の主な原因

以下、本書ではサイクル・アカウンティングの総和則を構成するの 5 つのコンポーネント（イベント）について、それぞれ専用の章を設けて説明を行う。説明の順番は、実際にアプリケーションの実行効率低下の原因として最も可能性の高いものから順に行っている。通常はメモリ・アクセスによるストールが最も一般的であるため、第 6 章と第 7 章ではそれぞれ BE_EXE_Bubble と BE_L1D_FPU_Bubble を取り上げ、この種のストールに関する問題について解説する。以降、第 8 章ではパイプライン・フラッシュ、第 9 章ではレジスタ・スタック・エンジン（RSE アクティビティ）、第 10 章ではフロントエンドにそれぞれ起因するストールについて見ていく。

データ収集に関する制限事項

第 4 章で述べたとおり、VTune アナライザでは 1 回のサンプリング実行で最大 4 つのイベントに関するデータを収集できる。これは、イベントの発生に関連した信号をカウントできる PMU（Programable Monitoring Unit）が 4 つ用意されているためである。このほかにも、同時に収集できるイベントの組み合わせについていくつかの制限事項がある。

L1 データ・キャッシュ・イベントに関する制限事項

ハードウェアの制約上、一部の L1-D キャッシュ・イベントは互いにラインを共有しているほか、他のイベントともラインを共有している。この結果、同時には使用できないイベントの組み合わせがあり、これらは別々にサンプリングを実行して使用する必要がある。L1-D キャッシュ・イベントは 5 つのセットに分割されている。1 回のデータ収集実行で計測できるのは、同じセット内のイベントのみである。異なるセットに属するイベントは同時には計測できない。例えば、L1DTLB_Transfer イベントと L2DTLB_Misses イベント（DTLB ミスの頻度をカウントする）はいずれもセット 0 に属している。一方、DLTB ミスによって失われたサイクルをカウントする BE_L1D_FPU_Bubble イベントはセット 1 に属しているため、セット 0 の DLTB イベントとあわせて 1 回のサンプリング実行でデータを収集することはできない。VTune アナライザは L1-D キャッシュ・イベントがどのセットに属しているかを認識し、サンプリングを必要な回数だけ自動的に実行する。L1-D イベントのデータ収集に関して、umask の値には一切制約はない。セット 0 およびセット 1 の L1D_READ イベントと DATA_REFERENCES イベントは機能的には同じものであり、セット0またはセット1イベントとそれぞれ組み合わせて使用できる。

以下に、L1-D キャッシュ・イベントの 5 つのセットを示す。

表 5-2　セット 0 イベント

イベント名	説明
L1DTLB_TRANSFER	L1D_READS でカウントされたアクセスで、L2DTLB でヒットした L1DTLB ミス
L2DTLB_MISSES	L2DTLB ミス
L1D_READS_SET0	L1 データ・キャッシュ読み出し
DATA_REFERENCES_SET0	メモリ・パイプラインに発行されたデータ・メモリ参照

表 5-3　セット 1 イベント

イベント名	説明
L1D_READS_SET1	L1 データ・キャッシュ読み出し
DATA_REFERENCES_SET1	メモリ・パイプラインに発行されたデータ・メモリ参照
L1D_READ_MISSES	L1 データ・キャッシュ読み出しミス

表 5-4　セット 2 イベント

イベント名	説明
BE_L1D_FPU_Bubble	L1 データ・キャッシュまたは FPU マイクロパイプラインに起因するパイプラインのストール・サイクル累計

表 5-5　セット 3 イベント

イベント名	説明
LOADS_RETIRED	リタイアしたロード
MISALIGNED_LOADS_RETIRED	リタイアしたアライメントの合っていないロード命令
UC_LOADS_RETIRED	リタイアしたキャッシュ不可ロード

表 5-6　セット 4 イベント

イベント名	説明
MISALIGNED_STORES_RETIRED	リタイアしたアライメントの合っていないストア命令
STORES_RETIRED	リタイアしたストア
UC_STORES_RETIRED	リタイアしたキャッシュ不可ストア

L2 データ・キャッシュ・イベントに関する制限

L2 キャッシュ・イベントは 6 つのセットに分割されている。同時に計測できるのは、同じセット内のイベント（または L2 以外のイベント）のみである。各セットは、PMC4 に設定されたイベント・コードによって選択される（すなわち、このセット内のイベントを計測したい場合、そのうちの 1 つは PMD4 で計測しなければならない）。セット内には、PMD4 でしか計測できないイベントもある。また、特定の同類の L2 イベントの umask を 1 次イベント（PMD4 を使用する L2 イベント）が指示するような umask については、いくつかの制限が存在する場合がある。これらの制限事項については、各セットの一覧の部分で説明する。各セットに属するモニタについては、次の節で詳しく説明する。

L2 キャッシュ・イベント（セット 0）

L2_OZQ_CANCELS* イベントまたは L2_IFET_CANCELS イベントのいずれか 1 つは PMD4 で計測しなければならない。これらのイベントは同じ umask を使用する。一度に計測できるのは、3 つある L2_OZQ_CANCELS* イベントのうち 1 つだけである。

表 5-7　L2 キャッシュ・セット 0 のパフォーマンス・モニタ

シンボル名	イベント・コード	最大インクリメント/サイクル	説明
L2_IFET_CANCELS	0xa1,0xa5,0xa9,0xad	1	L2 による命令フェッチのキャンセル
L2_OZQ_ACQUIRE	0xa2,0xa6,0xaa,0xae	1	L2 OZQ 内に存在する獲得順序属性に関するクロック
L2_OZQ_CANCELS0	0xa0	4	L2 OZQ のキャンセル
L2_OZQ_CANCELS1	0xac	4	L2 OZQ のキャンセル
L2_OZQ_CANCELS2	0xa8	4	L2 OZQ のキャンセル
L2_OZQ_RELEASE	0xa3,0xa7,0xab,0xaf	1	L2 OZQ 内に存在する解放順序属性に関するクロック

L2 キャッシュ・イベント（セット 1）

L2_L3 アクセス _CANCEL は、PMD4 で計測しなければならない。

表 5-8　L2 キャッシュ・セット 1 のパフォーマンス・モニタ

シンボル名	イベント・コード	最大インクリメント/サイクル	説明
L2_DATA_REFERENCE S	0xb2	4	L2 へのデータ読み出し / 書き込みアクセス
L2_L3 アクセス _CANCE L	0xb0	1	キャンセルされた L3 アクセス
L2_REFERENCES	0xb1	4	L2 から発行された要求

L2 キャッシュ・イベント（セット 2）

L2_FORCE_RECIRC は、PMD4 で計測しなければならない。

表 5-9　L2 キャッシュ・セット 2 のパフォーマンス・モニタ

シンボル名	イベント・コード	最大インクリメント/サイクル	説明
L2_FORCE_RECIRC	0xb4	4	強制された再循環
L2_ISSUED_RECIRC_OZQ_ ACC	0xb5	1	再循環発行を試行したが、先取権が与えられなかった回数
L2_GOT_RECIRC_OZQ_AC C	0xb6	1	OZQ アクセスが L1-D に再循環された回数
L2_SYNTH_PROBE	0xb7	1	合成されたプローブ

L2 キャッシュ・イベント（セット 3）

L2_Bad_Lines_Selected, L2_Bypass と L2_Store_Hit_Shared は同じ umask を共有する。

表 5-10　L2 キャッシュ・セット 3 のパフォーマンス・モニタ

シンボル名	イベント・コード	最大インクリメント/サイクル	説明
L2_BAD_LINES_SELECTED	0xb9	4	無効なラインが使用可能なときに有効なラインが置き換えられた場合
L2_BYPASS	0xb8	1	バイパスをカウントする
L2_STORE_HIT_SHARED	0xba	2	ストアが共有ラインをヒットした場合

L2 キャッシュ・イベント（セット 4）

L2_OPS_Issued、L2_Issued_RECIRC_IFETCH、L2_GOT_RECIRC_IFETCH のうちいずれか 1 つは PMD4 で計測しなければならない。これら 3 つのイベントは、同じ umask を共有する。

表 5-11　L2 キャッシュ・セット 4 のパフォーマンス・モニタ

シンボル名	イベント・コード	最大インクリメント/サイクル	説明
L2_GOT_RECIRC_IFETCH	0xba	1	命令フェッチ再循環が L2D によって受け取られた場合
L2_ISSUED_RECIRC_IFETCH	0xb9	1	命令フェッチ再循環が L2D によって発行された場合
L2_OPS_ISSUED	0xb8	4	L2D によって別の命令が発行された場合

L2 キャッシュ・イベント（セット 5）

L2_OZQ_FULL、L2_OZDB_FULL、L2_VICTIMB_FULL、L2_FILLB_FULL のうちいずれか 1 つは PMD4 で計測しなければならない。これら 4 つのイベントは同じ umask を共有する。

表 5-12　L2 キャッシュ・セット 5 のパフォーマンス・モニタ

シンボル名	イベント・コード	最大インクリメント/サイクル	説明
L2_OZQ_FULL	0xbc	1	L2D OZQ がフルになった場合
L2_OZDB_FULL	0xbd	1	L2D OZ データ・バッファがフルになった場合
L2_VICTIMB_FULL	0xbe	1	L2D ビクティム・バッファがフルになった場合
L2_FILLB_FULL	0xbf	1	L2D フィル・バッファがフルになった場合

BE_EXE_Bubble：メモリ・アクセスに起因するEXE パイプライン・ステージでのストール

6

アプリケーション実行時にパイプラインの EXE ステージでストール・サイクルが発生するのは、ほとんどの場合、実行ユニットが演算を行おうとした時点で必要なデータがレジスタに読み込まれていないのが原因である。こうしたデータ依存性によるストールは、以下のような理由によって発生する。

- メモリ・サブシステム（キャッシュまたはメイン・メモリ）からまだデータが到着していない。これをメモリ・アクセスによるストールと呼ぶ。
- 先行する機能ユニットの実行結果の書き込みがまだ完了していない。これは、先行する機能ユニットが演算を行ってから結果を書き込むまでのレイテンシが、命令のスケジューリングによって完全に吸収できていないのが原因である。これを機能ユニットのレイテンシによるストールと呼ぶ。

コンパイラは、これらのストールがなるべく発生しないように命令のスケジューリングを行う。しかしキャッシュやメモリのサイズには限りがあるため、コンパイラの意図したとおりにデータがキャッシュ・システム内に存在するとは限らない。また、ソースの一部に大量の演算を行う部分があれば、機能ユニットのレイテンシを隠蔽できないこともある。

本章では、以下の内容について解説する。

- パイプラインの EXE ステージにおけるストールの原因
- ストールに関係するアーキテクチャ機能
- これらストールの原因および相対的な重要性の見きわめ方

以上の内容を理解すれば、パイプライン・ストールの原因となっているパフォーマンス・ボトルネックを解消するための最適化ストラテジの構築と体系的な評価が可能となる。

メモリ・アクセスによるストール

メモリ・アクセスによるストールは、期待したキャッシュ内にデータが存在しない場合に発生する。このデータに依存している命令は、データのロードが完了して利用可能になるまでの間、ストールする。データが利用できない理由は主に 2 つある。

- データが目的の階層のキャッシュ内に存在しない（キャッシュ・ミス）
- 仮想アドレスから物理アドレスへの変換が最適に行われていない（DTLB ミス）

DTLB ミスがあると、アドレス変換完了後にキャッシュ・ミスが発生する。整数データを L1-D キャッシュから取り出すには、レベル 1 DTLB にヒットしなければならない。また、浮動小数点データを L2 キャッシュから取り出すにはレベル 2 DTLB にヒットしなければならない。このような種類のメモリ・アクセスによるストールは、BE_EXE_BUBBLE および BE_L1D_FPU_BUBBLE でカウントされる。これについては、第 7 章で詳しく解説する。

コンパイラが命令のスケジューリングを行う際は、以下のようにデータのロードが最適な条件で行われることを仮定している。

- 整数データは L1 キャッシュから 1 サイクルのレイテンシでロードする
- 浮動小数点データは L2 キャッシュから6サイクルのレイテンシでロードする

コンパイラは、命令のスケジューリングによってレイテンシを吸収しようとする。このとき、上記の最小値よりも多くのレイテンシを吸収することも可能ではあるが、少なくとも上記の最小レイテンシは吸収できるようにスケジューリングを行うのが普通である。

コンパイラがスケジューリングの際に前提とする最小レイテンシよりも実際のデータ読み込みのクロック・サイクルが大きくなる原因としては、基本的に以下の 2 つがある。

キャッシュ・ミス

期待したキャッシュ内にデータが存在しない場合に発生する。キャッシュ・ミスがあると、より低速なキャッシュやメモリ、あるいはディスクにアクセスしなければデータを取り出すことができない。

データ・アドレスの競合

1 サイクルで（あるいは少ないサイクルで連続的に）複数のデータ・アクセス命令が発行された場合に発生するストール。アドレスの競合が起こると、複数のデータ・アクセス間で相互に干渉が発生する。整数データと浮動小数点（FP）データではアクセス経路が異なるため、アドレス競合の仕組みも異なる。

メモリ・アクセスによるストールを解消するための最適化

最適化を行う際は、ストール・サイクルの要因として最も大きいものから重点的に改善することが重要である。全体的なパフォーマンスにほとんど影響しない部分を最適化しても無駄である。メモリ・アクセスによるストール・サイクルの削減に取り組む前にも、まずその問題がプログラムのパフォーマンスに重大な影響を与えているのかどうかを判断する必要がある。

以下、メモリ・アクセスによるストールを解消するための一般論を紹介する。ここで紹介する最適化手法は、ごく一部を除き特定のコンピュータ・アーキテクチャには依存しないものばかりである。本章および第7章「BE_L1D_FPU_Bubble : L1-D および FPU マイクロパイプラインに起因するストール」では、主にメモリ・アクセスによるストールの原因を特定する作業について解説する。ストールの性質がどのようなものかを正確に把握して、適切な対策を立てられるようになる。

キャッシュ・ミス

一般に、キャッシュ・ミスによって非常に多くのストール・サイクルが発生している場合は、データをあらかじめプリフェッチしてキャッシュ内に用意しておくか、使用するデータの局所性を高める必要がある。具体的には以下のような方法をとる。

- コンパイラの最適化レベルを高める。
- ソースの適切な場所にプリフェッチ組み込み関数を挿入する。
- 同一のデータに対してなるべく多くの処理を行うようにアルゴリズムの構造を変える。
- 連続したアドレス（すなわち、同じキャッシュ・ライン）に順次アクセスするようにデータ構造を変更する。キャッシュ・ミスが起こるのは、データを繰り返し使用する際の効率が悪いか、あるいはキャッシュ・ラインの概念を有効に活かしたデータ構造になっていないかのどちらかである。

よくあるのは、データを構造体の連結リストとして構成しているケースである。この場合、プログラムが連結リストをたどってデータを取り出す必要がある。この場合に最も効果的と考えられるのは、構造体のブロックに対してスペースを割り当て、このブロックの中でデータを配列構造とすることである。こうすると構造体の要素を連続的に格納できるため、キャッシュ・ラインの使用効率が高まる。しかも、現在のブロックを分析している間に連結リストの次のブロックをプリフェッチできるため、メモリ・アクセスのレイテンシを完全に隠蔽できる。

一般に、構造体の中のデータをアルゴリズムが一度にたくさん使用することはあまりない。これではキャッシュ・ラインに含まれているデータのごく一部しか使用されなくなり、キャッシュをライン単位でロードする意味がほとんどなくなってしまう。

キャッシュを効率的に使用する観点から見れば、構造体の連結リストよりも配列構造を使用した方がよい。実際にこれが難しい場合は、浮動小数点データと整数データを別々の並列なデータ構造に格納する。これによって、L1（整数の場合のみ）および L2 キャッシュのキャッシュ・ライン構造をより効果的に利用できるようになる。

あるいは、コードがある特定のデータ・ブロックを何度も使用しているにもかかわらず、その間にあまり使用されない大きなデータ・ブロックを読み込んでいる場合は、目的のデータ・ブロックが必要になるたびにキャッシュにロードし直さなければならない。このような場合、もし可能であれば、同じデータを使用する作業を完全に終わらせてから別の作業に移るようにアルゴリズムの構造を変更する。このように、キャッシュ・ラインの概念の意味を十分に理解した上でアルゴリズムを構築することが重要である。

また、データが高速キャッシュには存在せず、L3 キャッシュに格納されている場合には、これ以外にも利用できる手法がある。ループ・コードの場合、メモリ・アクセスのレイテンシはパイプライン化およびレジスタのローテートによって吸収される。ループのアンロールを積極的に行うと、ループの 1 イタレーション当たりのサイクル数を増やし、パイプライン化によってより多くのレイテンシを吸収できるようになる。これを積極的に進めていって、コンパイラによる最適化が L3 レイテンシ（13 サイクル。詳しくは後述）を自動的に吸収できるようにすれば、キャッシュ・ミスによるパイプライン・ストールは発生しなくなる。Itanium® プロセッサ・ファミリ・アーキテクチャ上で動作するアプリケーションを最適化する際は、この手法も選択肢として考えておくとよい。

アドレスの競合

アドレスの競合は一般に考えられているよりも頻繁に発生するものであり、キャッシュ・ミスに匹敵するペナルティをもたらす。アドレスの競合には、キャッシュおよびキャッシュ・アクセス・ハードウェアのきわめて詳細なメカニズムが関係しているため、原因を理解するのは難しいが、キャッシュ・ミスよりは簡単な方法で回避できる。

アクセスの競合を発見するには、VTune™ アナライザのイベント・ベースのサンプリング（EBS）機能を利用する。そして、アドレスの競合を発見したらその場所を特定し、次にデバッガを使ってアドレスの競合を起こしている変数を特定する。

アドレスの競合が起こるのは、単独のキャッシュ、または複数のキャッシュが使用しているアクセス・メカニズムの仕組みに原因がある。Itanium® 2 プロセッサの場合、アドレスの競合には大きく 2 種類がある。1 つはバンク競合、そしてもう 1 つは同一の L2 キャッシュ・ラインに対するセカンダリ・ミスである。

バンク競合

L2 キャッシュは 16 バイトのバンク 16 個で構成される。特に浮動小数点ロードの場合によく見られるように、1 サイクルで 2 つのロードが同じバンクにアクセスしようとすると、一方のアクセスが強制的に L2 OzQ によって再発行される。すると、再発行されたロードが浮動小数点データにアクセスする際のレイテンシは通常の 6 サイクル（2 つのアクセスが別々のバンクに対して行われた場合の値）から 12 サイクルに増えてしまう。

バンク競合を引き起こすメカニズムには、次の 2 つがある。

- 同一のキャッシュ・ライン内に含まれる 2 つの要素間でのアクセス競合。
- 2 つのデータ・ブロック（配列）が同一のベース・アドレス・アライメントおよびコヒーレントなアクセス・パターンを有している場合のアクセス競合。

このうち、後者はループ・コードが複数の配列にアクセスするような場合に特に大きな問題となる。しかし、これらのアライメントは容易にチェックが可能である。

構造体の連結リストがアライメントの問題を引き起こしている場合は、すべてのアクセスで毎回この問題が起こるとは限らないため、対処はやや困難である。特に、連結リストを動的に管理している場合は、各要素のアライメントはきわめて複雑な作業となる。これも、データ構造として連結リストを使うことの短所といえる。この種のコーディング手法はエレガントでありスペースも節約できるが、データを効率的に読み込むコードをコンパイラで生成するのがきわめて難しくなるため、ハードウェアが機能ユニットに対してデータのタイムリーな供給がよりいっそう難しくなる。

アドレスの競合の例として、Itanium® 2 プロセッサでの浮動小数点アクセスを考えてみる。浮動小数点データは直接 L2 キャッシュから浮動小数点レジスタ・ファイルにロードされる。同じバンクにアドレスがマッピングされる 2 つの配列があり、この配列に対する 2 つのロード命令が 1 クロック・サイクルで同時に発行されたとする。この場合、これらロード命令によって生成される OzQ エントリのうち一方は、再発行される。これによって、2 つのロードを完了するまでのレイテンシは、OzQ エントリの再発行による影響を受けるため、通常の 6 サイクル（2 つのアクセスが別々のバンクに対して行われた場合の値）から 12 サイクルに増える。

この種のアドレス競合は、その存在さえ特定できればきわめて簡単に解決できる。その例を、以下に示す。

例：

次のコード行は 256 バイトにアライメントされたバッファを作成する。

```
buf = buf + 256 - ((UINT64)buf%256);
```

ただし、UINT64 は typedef で 64 ビット符号なし整数と宣言されているものとする。すなわち、ポインタである。

この場合、アドレスは 16 バイトの倍数ごとに増えていき、バンク競合は解消される。なお、2 つのロードを同時に行えるようにするため、バッファを常に 16 バイト境界でアライメントしておく必要がある。malloc は 16 バイトにアライメントされたアドレスを返す。

実際のアプリケーションでは、キャッシュ・ラインの入れ替えが定期的に発生する。この入れ替えによって、アドレスとバンク割り当ての対応づけは毎回変化する。L2 キャッシュでは 16 バンクが 2 本の L2 キャッシュ・ラインに対応する。これを 4 つ組み合わせて、8 ウェイ・アソシアティブ・セットを 1 つ構成する。

本書で紹介するマイクロベンチマークはごく単純なアドレス競合を示したものである。したがって、こうした影響は必ずしも決まって現れるとは限らない。アプリケーションがアルゴリズム的に定義されている場合、すなわちコード実行のほとんどがデータの値に無関係に行われるような場合は、キャッシュ内での相対的なデータ・アライメントの問題は簡単に解決できる。一方、データの値に大きく依存するアプリケーションの場合は、キャッシュのバンク構造内におけるデータの相対的なアライメントを予測することが困難になる。

一般論として、あるソース行が BE_EXE_Bubble でカウントされるストール・サイクルを多発している場合は、そのソース行に関連したデータ・アクセスでバンク競合や再循環の発生イベントが発生していないか確認する必要がある。そのような症状が確認されたら、アプリケーションをデバッガにかけて変数（ソースおよびターゲット）のアドレスを調べるとよい。

Itanium 2 プロセッサには、アドレス競合を調べるための発生イベントが用意されている。VTune アナライザで L2_OzQ_Cancels1.Bank_Conflict イベントを使用すると、明示的にバンク競合をカウントできる。データを収集したら、ソースおよびアセンブリ・コードのレベルにまでドリルダウンしてバンク競合の発生個所を特定することができる。

セカンダリ L2 キャッシュ・ミス

すでに未処理のキャッシュ・ミス状態にあるキャッシュ・ラインに対して L2 キャッシュ・ミス（セカンダリ・ミス）が起こると、OzQ はデータ・アクセスを再循環する。L2 内でキャッシュ・ミスがあった場合、キャッシュ・ラインへの未処理のアクセスは、一度に 1 つしか L3 やシステム・バスにアクセスができない。そして、未処理のラインの更新中に発生した後続のキャッシュ・ミスはキャッシュ・ラインの更新が完了するまで再循環する。最初のミス（プライマリ・ミス）に関しては、L2 キャッシュにキャッシュ・ラインの最初の部分がロードされた時点でデータを返されるが、キャッシュ・ラインの更新が完了する前に発生したセカンダリ・ミスについては、キャッシュ・ラインが完全に更新されるまで待たなければならない。このため、データをレジスタに格納するまでのレイテンシはさらに大きくなる。

このようなレイテンシは、いくつかの方法で解決できる。最も効果的なのは、以下のいずれかの方法でキャッシュ・ミスのレイテンシを完全に回避するものである。

- 最適化オプション /O3 をつけてプログラムをビルドして、プリフェッチを生成する。
- プリフェッチ組み込み関数を使用する。

このほかにも、セカンダリ・ミスは以下の方法で回避できる。

- 同一の構造体または配列にアクセスしているソース行どうしを十分に離し、最初のアクセスによるキャッシュ・ラインの更新が完了してから 2 回目以降のアクセスを行うようにする。データが L3 でヒットすると仮定した場合、これには約 10 サイクルを要する。
  注意 : データがメイン・メモリにある場合にはこの手法は利用できない。
- あるいは、構造体の配列（または連結リスト）を配列の構造体で置き換えれば、これらの要素が同一のキャッシュ・ラインに格納されるのを避ける。

通常、Itanium® 2 プロセッサではセカンダリ L2 ミスによる再循環を正確にカウントできないが、L2_FORCE_RECIRC の 4 つのサブイベントを合計すると、再循環の上限値は知ることができる。しかしこの合計はセカンダリ・ミスの一部を二重にカウントしていたり、あるいは同一のアソシアティブ・セットの複数のキャッシュ・ラインに対するミスもカウントしている場合がある。これについては、本章の最後で詳しく解説する。

機能ユニットのストール

コンピュータでは、計算結果が正しいことを確認するために機能ユニットのレイテンシによるストールが必ず必要となる。命令チェーンでは、1 つの命令の出力が別の命令の入力として使われるときがある。この場合、結果を出力する側の命令とその結果を使用する側の命令の間に十分な数の命令（サイクル）がなく、データ生成に要するレイテンシを吸収できなければ、後続の命令はデータの準備ができるまでストールしなければならない。これを「スコアボード」ストールと呼ぶ。Itanium® 2 プロセッサでは、ほとんどの整数演算を 1 サイクルのレイテンシで実行できる。RAW（Read-after-Write）の依存性違反を避けるには、結果を出力する側の命令とその結果を使用する側の命令の間に 1 サイクルが必要である。つまり、実行レイテンシが 1 サイクルの整数命令が機能ユニットのストールを引き起こすことはない。機能ユニットにストール・サイクルを発生させるのは、通常はマルチメディア（MM）整数命令や浮動小数点演算である。

整数命令の場合、機能ユニットのストールは BE_EXE_Bubble.GRGR イベントによって明示的にカウントされる。このイベントは、BE_EXE_Bubble.GRALL がカウントするストール・サイクルの一部をカウントする。このため、機能ユニットのストールが実行効率の低下の原因となっているかどうかは簡単に調べられる。通常は、組み込み関数を使ってコーディングした MM 命令がチェーンを形成している場合に機能ユニットがストールすることが多い。これを解決するには、MM 命令の間に他の命令をインターリーブさせて、レイテンシを吸収する手段をとる。

浮動小数点演算を多用するアプリケーションでは、機能ユニットのストールが発生しやすい。これは、複雑な計算を構成する基本的な浮動小数点命令は一般的に実行レイテンシが大きいためである。また、整数演算の場合に比べ、浮動小数点演算では機能ユニットのストールは他のストールとの区別がつきにくい。BE_EXE_Bubble.FRALL は浮動小数点メモリ・アクセスによるストールと機能ユニットのレイテンシによるストールの両方をカウントするが、これらを区別してカウントするための FRFR サブイベントが用意されていないためである。実際にどのような状況が発生しているのかを調べるには、VTune™ アナライザのソース・ビューでコードのディスアセンブリ・リストを表示して確認するとよい。コードを分析した後でもなお機能ユニットのレイテンシが問題となる場合は、レイテンシを吸収できるように演算の配置を工夫する必要がある。具体的には、中間結果を集め、数式のグループ化を変更して、除算および平方根の計算（この演算は特にレイテンシが大きい）の数を減らすようにする。このような場合、ルックアップ・テーブルや補間計算を使用しても十分な計算精度が維持できるなら、パフォーマンスの改善には最も効果的な方法であるといえる。このストラテジを用いると、ストール・サイクルの削減に加え、パフォーマンスも大幅に改善できる。ただし、このストラテジにはスペースとスピードのトレードオフが存在する。

BE_EXE Bubble

メモリ・アクセスによるストール・サイクルのほとんどは BE_EXE_Bubble カウンタでカウントされる。このカウンタは、パイプラインの EXE ステージで発生するストール・サイクルをカウントする。このようなストール・サイクルが発生するのは、主に機能ユニットが必要とするデータのレジスタへのロードが完了していないためである。このような依存性によるストールは、その原因によって次の 2 つに分類できる。

- 機能ユニットがデータを必要とした時点でデータがまだメモリ・サブシステムから読み込まれていない場合
- 機能ユニットからデータがレジスタに書き戻されておらず、後続の命令でそのデータを使用できない場合

上記いずれの場合も、レジスタにデータを格納する命令とそのデータを使用する後続命令との間に、依存関係のない命令を十分に置いていないために、レジスタにデータを読み込む際のレイテンシを完全に吸収できていないことが原因である。

BE_EXE_Bubble カウンタのサブイベントを分析すれば、パイプラインのストールを引き起こすこの種のメモリ・アクセスについて明確な理解が得られる。

以下に、BE_EXE_Bubble のサブイベントの一覧を示す。これらのサブイベントは umask の値を変えれば選択できる。VTune™ アナライザでは、サブイベントは特定の umask 値であらかじめ定義されている。

表 6-1　BE_EXE_Bubble のサブイベント

拡張子	PMC.umask	説明
ALL	B0000	exe によってバックエンドがストールした場合
GRALL	B0001	GR/GR または GR/ ロードの依存関係が原因で、バックエンドが exe によってストールした場合
FRALL	B0010	FR/FR または FR/ ロードの依存関係が原因で、バックエンドが exe によってストールした場合
PR	B0011	PR の依存関係が原因で、バックエンドが exe によってストールした場合
ARCR	B0100	AR または CR の依存関係が原因で、バックエンドが exe によってストールした場合
GRGR	B0101	GR/GR の依存関係が原因で、バックエンドが exe によってストールした場合
CANCEL	B0110	キャンセルされたロードが原因で、バックエンドが exe によってストールした場合
BANK_SWITCH	B0111	バンク切り替えが原因で、バックエンドが exe によってストールした場合
ARCR_PR_CANCEL_BANK	B1000	ARCR、PR、CANCEL、または BANK_SWITCH
---	B1001-b1111	(* 何もカウントされない *)

汎用レジスタについては、2 種類の依存性によるストールを明示的に計測することができる。

- BE_EXE_Bubble.GRALL：整数データの依存性によるストールをすべてカウントする。
- BE_EXE_Bubble.GRGR：整数機能ユニットのレイテンシが命令スケジューリングによって完全に吸収されていないために発生するストール・サイクルをカウントする。

これらのサブイベントを利用すると、整数データのロードに起因するストール・サイクルの概算値は次式で求められる。

```
BE_EXE_Bubble.GRALL- BE_EXE_Bubble.GRGR
```

一般に、コンパイラは命令のスケジューリングによって機能ユニットのレイテンシをほぼ隠蔽するため、通常は BE_EXE_Bubble.GRALL でカウントされるデータのみを収集すれば、整数データのロードに起因するストール・サイクルの概算値を得られる。なお、これら 2 つのカウンタには優先度は設定されておらず、1 サイクルのストールを両方のカウンタが計測する場合もあるため、上記の減算では正確な値を求めることはできない。例えば、ロード命令と MM 命令が同じサイクルで発行されたとして、ロードが L1 にヒットしなかったとする。ここで、十分な間隔を空けずにこれら 2 つの命令の結果を後続のコードが使用しようとすると、メモリ・アクセスによるストールと機能ユニットのレイテンシによるストールが両方発生するが、実際には後者が前者を隠蔽してしまう。

浮動小数点ロードには、このようなサブイベントのペアが用意されていないため、浮動小数点データ・アクセスに起因するストール・サイクルの計測は厳密には行えない。パフォーマンス・イベントのみを使用するのであれば、関連するメモリ・サブシステムの発生イベント（BE_EXE_Bubble.Frall 以外）を観察して、浮動小数点データ・アクセスによってどれだけ重大なストールが発生しているかを調べる必要がある。VTune アナライザでディスアセンブリ・ビューにドリルダウンしてコードの検証を行えばこれらの点を調べられる。

ここからは、BE_EXE_Bubble.GRALL-BE_EXE_Bubble.GRGR、BE_EXE_Bubble.FRALL でカウントされるメモリ・アクセスのストール・サイクルについて解説する。EXE ステージでカウントされるストール・サイクルのほとんどはこれらの要素が原因と考えられる。もちろん、EXE ステージではコンパイラによるスケジューリングで吸収しきれなかった機能ユニットのレイテンシ（例：整数機能ユニットのレイテンシをカウントする BE_EXE_Bubble.GRGR など）によるストール・サイクルもカウントされる。機能ユニットのレイテンシがストール・サイクル全体（または CPI）に対して大きな割合を占めている場合は、メモリ・アクセスを行っているコードどうしの間にたくさんの演算コードをインターリーブする必要がある。こうすれば、コンパイラはより柔軟にスケジューリングが行え、ストール・サイクルは低減するはずである。しかし、実際に行うのはきわめて難しいときがある。

メモリ・アクセスによるレイテンシのペナルティ

メモリ・アクセスのストールにはいくつもの種類があり、その相対的な重要性を評価するには、それぞれの種類のストールのペナルティについて理解しておく必要がある。メモリ・アクセスでストールが発生する主な原因としては、以下のケースがある。

- キャッシュ・ミス：この場合、階層の低い（レイテンシの大きい）メモリ・サブシステムからデータを取得する。通常、コンパイラは最も高速なキャッシュにデータがあるものと仮定してスケジューリングを行うため、レイテンシの大きいメモリ・サブシステムにアクセスすると EXE ステージでストールが発生する。
- DTLB ミス（次章で解説）
- アドレスの競合：データ・アクセス・ハードウェアがより複雑なアクセス・パターンを実行するため、データ・アクセスの実効レイテンシが大きくなる。

以上の点を理解しておけば、メモリ・アクセスによるストールの内訳の相対的な影響を評価できる。つまり、これらのストールに関連した発生イベントの重要性を、それぞれのペナルティの大きさによって判断し、サイクル・アカウンティングを近似的に再構築する方法をとる。L2 キャッシュからはデータがアウトオブオーダーに返されるため、このようなサイクル・アカウンティング・モデルは、Itanium® プロセッサのときほど厳密には行えない。しかも、コンパイラがスケジューリングによって吸収するレイテンシの量についても仮定に頼らざるを得ない。したがって、本章で紹介する概算値の関係式は、各発生イベントに関係するストール・サイクルの上限値を知るための参考のみにとどめる。こうした制約はあるものの、このモデルは実行効率低下の原因を評価するためのガイドとしては十分役に立つ。

実際のアプリケーションにおけるメモリ・アクセスのペナルティ

実際のアプリケーションにおけるメモリ・アクセスは、マイクロベンチマークで観察したものよりもはるかに複雑である。マイクロベンチマークは本質的に微細なものであり、データ・アクセスの細かいコーディング方法による影響を受けやすい。本書で紹介するマイクロベンチマークはいずれも単一のアーキテクチャ機能のみを対象としており、シーケンシャルなスケジューリングによって明示的に検証を行っている。しかし実際のアプリケーションでは、コンパイラがレイテンシの大きい操作を可能な限り並列にオーバーレイするため、レイテンシが隠蔽される。また、Itanium® 2 プロセッサでは L2 および L3 キャッシュからのデータもアウト・オブ・オーダーに返されるため、さらに複雑になっている。このため、ごく単純なペナルティを利用してストール・サイクルの内訳を判断しようというのは、数値的にあまり正確なものにはなり得ない。それでも、どのメモリ・アクセスの問題から解決していくべきか、その相対的な重要性を判断する意味では、このアプローチは十分に有効なガイドとして機能する。

メモリ・アクセスのペナルティの測定

Itanium® 2 プロセッサのキャッシュはデータをアウトオブオーダーに返せるため、高級言語で記述した 1 つの命令がメモリの読み出しや書き込みに対する要求をいくつも生成すると、メモリ・アクセスのストールは複雑になってくる。Itanium 2 プロセッサのアクセス・ストール・カウンタを解釈するには、以下の手順をとる。

- まず、1 つの変数のみを使用した最もシンプルなアクセスについて観察する。配列のサイズをさまざまに変更することによって、どのメモリ・サブシステム（キャッシュまたはメイン・メモリ）にアクセスするかを強制的に指定する。
- 次に、複数のデータ・アクセスを分析して、アドレスの競合がデータ・アクセスの効率をどのように低下させているかを調べる。

いずれの場合にも、本書ではソース・コードを修正してデータ・アクセスの方法を変更して、メモリ・アクセスによるストールを抑えるための一般的な推奨事項について説明する。

単純なキャッシュ・ミスに起因するメモリ・アクセスのストール

ロード命令によってデータ・アクセスが行われる際、目的のデータがデータ・キャッシュ・アレイに格納されていなければ、キャッシュ・ミスが発生する。整数ロードの場合は、まず最初に L1-D にアクセスする。浮動小数点ロードの場合は最初に L2 キャッシュにアクセスする。ロード命令は、目的のデータを見つけるまでキャッシュ階層を L1、L2、L3 の順（浮動小数点の場合は L2、L3 の順）にアクセスし、次にメモリ、そして最後にはディスクにアクセスする。コンパイラは、必要なデータが最も高速なキャッシュ・システムに格納されている前提でデータの使用に関するスケジューリングを行う。データ・アクセスの際にメモリ階層を下がっていくにしたがって、最終的にデータを読み込むまでのレイテンシも増加する。このとき、各メモリ階層のキャッシュ・ミスおよびキャッシュ参照のカウンタも加算されていくため、これらカウンタを調べればストールの詳しい原因を把握できる。

キャッシュ・ミスが発生するのは次のいずれかの場合である。

- 必要なデータを含むキャッシュ・ラインが他のデータで置き換えられた場合
- 必要なデータを含むキャッシュ・ラインがまだロードされていない場合

単純なアクセスに関するマイクロベンチマーク

最も高速なキャッシュにミスした場合に発生するメモリ・アクセスのストールのペナルティを調べるために、以下のマイクロベンチマークを使用する。

```
do_read_inner:
{ .mmi
add r29 = r30, r33;;
ld4 r28 = [r31]
and r30 = r29, r34;;
}
do_read_branch:
{ .mib
mov r29 = r28
add r31 = r30, r35

br.cloop.dptk.few do_read_inner;;
}
```

```
do_calibrate_inner:
{ .mmi
add r29 = r30, r33;;
nop.m 0
and r30 = r29, r34;;
}
do_calibrate_branch:
{ .mib
mov r29 = r27
add r31 = r30, r35

br.cloop.dptk.few do_calibrate_inner;;
}
```

r28 から r29 へデータを移動させると、ループを 1 回実行するたびにレイテンシを強制的に発生させている。このテストでは、アクセス先のバッファ範囲の調整により、ターゲットとなるキャッシュ階層を強制的に設定している。これを制御しているのが r34 の値である。比較対象として、ロード命令を実行していない点以外はすべて同じコードを右側に示した。

上記のコードを実行した結果を下の表に示す。ただしこれは上記のマイクロベンチマークの結果に過ぎず、このようなレイテンシが絶対的な結果として得られる意味ではない。これらの結果は、アプリケーション分析のガイドラインとなる関係式を得るためだけに使用する。

アクセス先のメモリ・サブシステム	レイテンシ（サイクル数）
L1	1
L2	5
L3	13.3
メモリ	209.6

上記の結果は、整数データ・アクセス時のペナルティの測定結果である。L3 レイテンシは『インテル® Itanium® 2 プロセッサ・リファレンス・マニュアル：ソフトウェアの開発と最適化』で解説されている値よりも若干大きくなっている。L3 レイテンシの最小値を観察するには、もう少し正確なテストを作成する必要があるが、これはあまり一般的ではない。本書では、上記の測定値を基準にして、他のテストとの比較を行う。また、メイン・メモリのレイテンシは使用するチップセットによっても異なる。本書で紹介しているレイテンシの値は、インテル 870 チップセット・ベースのインテル・プラットフォーム上で測定したものである。なお、この表で示している値はマイクロベンチマークの測定結果に 1 サイクルを加算している。これは、最初のバンドル内の 2 番目のストップ・ビットを発行するために 1 サイクルが必要なためである。

次に、上記のコードを一部修正し、浮動小数点データのロードを行うようにしたマイクロベンチマークを以下に示す。この場合、データがレジスタ・ファイルに到達するまでの経路が変わるため、ペナルティの値も若干変化する。浮動小数点データは L2 キャッシュから直接ロードされる。

```
do_read_inner:
{ .mmi
add r29 = r30, r33;;
ldfs f28 = [r31]
and r30 = r29, r34;;
}
do_read_branch:
{ .mib
mov f29 = f28
add r31 = r30, r35

br.cloop.dptk.few do_read_inner;;
}
```

```
do_calibrate_inner:
{ .mmi
add r29 = r30, r33;;
nop.m 0
and r30 = r29, r34;;
}
do_calibrate_branch:
{ .mib
mov r29 = r27
add r31 = r30, r35

br.cloop.dptk.few do_calibrate_inner;;
}
```

上記のコードを実行した場合のレイテンシを以下の表に示す。ここでも、最初のバンドル内の 2 番目のストップ・ビットの発行サイクルを考慮して、マイクロベンチマークの測定結果に 1 サイクルを加えている。

アクセス先のメモリ・サブシステム	レイテンシ（サイクル数）
L2	6
L3	13.1
メモリ	209.5

これらの結果をもとに、メモリ・アクセスによるストールを説明するモデルを構築できる。構築するモデルは、実行時の効率を低下させている主な原因を判断するためのガイドとしての利用のみを想定している。この結果から得られる関係式は、使用した発生イベントのカウント数に関連するストール・サイクルを常に多めに見積もってしまう点に注意が必要である。他のイベントが何も発生していないと仮定すれば、後述する近似式が成り立つ。しかし次章で説明するとおり、TLB ミスがある場合は、メモリ・アクセスによるストール・サイクルの要因に関する関係式はここで紹介するものよりもやや複雑になる。また、同時に複数のメモリ・アクセスが発生すると互いに隠蔽しあう点にも注意が必要である。

通常、コンパイラは最短のレイテンシを前提にスケジューリングを行っており、データをロードして使用するまでの間のレイテンシが最小であれば、完全に吸収できる。メモリ・アクセスによるストール・サイクルのうち、キャッシュ・ミスに起因する部分を計算で求めるには、コンパイラが前提としている最短のレイテンシの値を差し引けばよい。整数演算を多用するアプリケーションでは、以下の近似式が成り立つ（ここでは浮動小数点データのキャッシュ・ミスは考慮しない）。

```
BE_EXE_Bubble.GRALL - BE_EXE_Bubble.GRGR ~
    (L1_Read_Misses-L3_Reads.Data_Read.All)*(5-1) +
            L3_Reads.Data_Read.Hit*(13.3-1) +
    L3_Reads.Data_Read.Miss*(209.6-1)
```

浮動小数点を多用するアプリケーションでは、整数ロード時のキャッシュ・ミスは事実上無視できるため、BE_EXE_Bubble.FRALL の内訳について、次の近似式が成り立つ。

```
BE_EXE_Bubble.frall~
    L3_Reads.Data_Read.Hit*(13.1-5)
    L3_Reads.Data_Read.Miss*(209.5-5)
```

ここではコンパイラが最小のキャッシュ・レイテンシを前提にスケジューリングを行ったものとする。しかしこの式にはある種の難しさが伴う。というのは、OzQ は L3 およびシステム・バスに回した複数のキャッシュ・ミスを 1 つにまとめるためである。この結果、L2 にヒットした L1 ミスによる影響が多めに見積もられるようになり、1 つにまとめられたミスの再循環による影響はまだ含まれていない。これについては、同一のキャッシュ・ラインに対して複数のミスが発生した場合について解説した節で詳しく取り上げる。

次に、これらのコンポーネントをリタイアした命令数で正規化すると、CPI に対するおおよその影響を計算できる。

整数データを多用するコードでは、次式を計算する。

```
CPI(L2 Hits) = (L1_READ_MISSES-L3_Reads.Data_Read.All)
               *4/IA64IR
CPI(L3 Hits) = (L3_Reads.Data_Read.Hit*12.3/IA64IR
CPI(L3 Misses) = L3_Reads.Data_Read.Miss*208.6/IA64IR
```

浮動小数点データを多用するコードでは、次式を計算する。

```
CPI(L3 Hits) = L3_Reads.Data_Read.Hit*8.1/IA64IR
CPI(L3 Misses) = L3_Reads.Data_Read.Miss*204.5/IA64IR
```

上記の計算を行えば、プログラムの実行効率を低下させている主な要因がわかる。L1 キャッシュ・ミスは L3 キャッシュ・ミスよりもはるかに頻繁に発生するが、L3 キャッシュ・ミスのペナルティは L1 キャッシュ・ミスが L2 にヒットした場合に比べて 50 倍も大きい。

ここで、L2 データ・ミスの計算に L3_Reads.Data_Read.All イベントを使っている点に注意が必要である。L2_Misses イベントは書き込みに起因するキャッシュ・ミスもカウントする。キャッシュはライトスルーであるため、ミスしているキャッシュ・ラインに対して書き込みを行おうとすると、キャッシュ・ラインを取り出す必要があり、これによってキャッシュ・ミスが生じる。

キャッシュ・ミスによるストールの最適化

前節で示した式（整数または FP）が全体的な CPI の中で大きな割合を占めている場合は、その中でも特に大きな要因となっているキャッシュ・ミスを削減する必要がある。最も明快な解決方法は、すべてのループがメモリ内のデータを順番にアクセスし、キャッシュ・ラインの構造を効率よく利用しているかどうかを確認することである。C 言語の多次元配列の場合、ループは最後の添字にアクセスするようにする。Fortran の場合は最初の添字にアクセスするようにする。

最適化レベルを /O3 に引き上げると、HLO（High Level Optimizer）が実行される。そしてコンパイラ自身の判断によって適切なプリフェッチ命令が生成される。しかしプリフェッチ命令もキャッシュ・ミスを引き起こすため、これによってキャッシュ・ミスを減らすことはできない。しかしプリフェッチによって、通常よりも早い段階でバイナリがデータを要求できるようになるため、低速なメモリ・サブシステムにアクセスする際に余分に必要となるレイテンシは吸収できる。

すでにコンパイラの最適化レベルを最高にしている場合は、ia64intrin.h に含まれるプリフェッチ組み込み関数をソース・コードに使用すると、データを前もってフェッチしてアクセス・レイテンシを隠蔽できるようになる。

void __lfetch（int lfhint, void *y）

ここで、この組み込み関数は lfetch.lfhint 命令を生成する。1 番目の引数の値は、どのキャッシュ階層にデータをプリフェッチするかを決めるヒント・タイプを指定する。例えば、浮動小数点データは必ず L2 キャッシュにプリフェッチする必要がある。

__lfetch（nt1, address）

アクセスの順序づけが正しく行われているものとして、最も高速なキャッシュおよびキャッシュ階層全体を有効に利用するには、データ・アクセスの「ブロック化」を図る。これを体系的に行うには、以下のような手法を使用する。

- 問題のサイズを最も高速なキャッシュ（整数の場合は L1-D、浮動小数点の場合は L2）内に収まるように縮小する。
- データの一部を一時的な配列にコピーするモジュールを作成して、すべてのアクセスをシーケンシャルに行えるようにする。
- 内側のカーネルが最大のスピードで動作するようにチューニングする。

ここでは、大規模な問題をコピー・モジュールやカーネル・モジュールが扱えるサイズに分割する。そうすると、本体は適切なサイズのブロックによって 1 つのループとなる。

重要なのは、一時的な配列にデータをコピーすればシーケンシャルなアクセスを行える以外にも別の効果が得られることである。しかも、こちらの効果の方が重要性が高いことが多い。それは、キャッシュ・ラインを強制的に入れ替えできるからである。最悪のケースとして 4096 × 4096 の倍精度浮動小数点の行列があり、これを 32 × 32 のサブ行列にブロック化した場合を考える。これらのサブ行列は、サイズがキャッシュ・ライン 2 本分に相当し、短いシーケンシャルなセグメントが連続したものとしてメモリに格納される。これらのベース・アドレスは、配列の第 1 次元の大きさが 4096 であるため、32768 バイト（アドレス内の 15 ビット）離れていることになる。L2 は 128 バイト長のキャッシュ・ライン 8 本で 1 行を構成し、これが 256 行あるため、L2 キャッシュ内のアソシアティブ・セット（行）はアドレスのビット 8 からビット 15 までで決定される。この結果、32 × 32 のサブ行列は 256 行のアソシアティブ・セットのうち 2 つにしか格納できず、ある特定の瞬間に注目すると、L2 キャッシュ内に格納できるのはこのサブ行列の 1/4 に過ぎないことになる。こうした影響のため、すべてのデータがごく一部のアソシアティブ・セットに集中的に格納され、キャッシュ全体の利用効率が大きく低下するケースが実際問題として発生する場合がある。

BE_EXE_Bubble.GRALL - BE_EXE_Bubble.GRGR または BE_EXE_Bubble.FRALL の内訳として、キャッシュ・ミスによるペナルティが大きな割合を占めている場合でも、他の原因によってメモリ・アクセス・ストールが発生していないかどうかを調べる必要がある。これは、OzQ が複数のアクセスのオーバーラップをサポートしているためである。次節では、キャッシュ・ミスに次いでストールを引き起こす可能性が高いバンク / アドレス競合について解説する。

バンク / アドレスの競合、EXE ストール・サイクルに対する影響

メモリ・アクセスによるストール・サイクルが顕著であるにもかかわらず、前節で説明した計算式で原因が特定できない場合は、ストール・サイクルの別の原因を考える必要がある。多くの場合、マイクロプロセッサのキャッシュ・アーキテクチャ構造はデータ・アクセスのレイテンシをきわめて低く抑える働きをするが、必ずしもその働きが機能しないときもある。例えば複数のデータを同時にロードする必要のあるプログラムでは、コンパイラは同一のサイクルで 2 つのロード命令を発行するコードを生成しなければならない。このような場合にアドレスの競合が発生し、データ・アクセスのレイテンシが大きくなることがある。これは、データ・サンプルのサイズが大きい浮動小数点データを多用するアプリケーションで起こりやすいが、もちろん整数データへのアクセス時にもアドレスの競合は発生する。以下、これら両方の場合について解説する。

L2 キャッシュは 256KB、8 ウェイ・セット・アソシアティブ、そしてキャッシュ・ラインの長さは 128 バイトである。このキャッシュは 16 バイト幅のバンク 16 個で構成されている。同一サイクルで 2 つのロード命令が発行され、これらのデータが L2 内でアドレス競合またはバンクが競合の状態にあると、複雑な事態が発生する。両方のロードが L1 でヒットすれば競合は発生せず、両方のロードが 1 サイクルで完了する。

同一の L2 バンクに格納されているデータに対して複数のメモリ・アクセスが実行されると、競合が発生する。この場合、L2 OzQ によって通常とは異なるアクセス・メカニズムが呼び出されるため、レイテンシが大きくなる。浮動小数点データの場合は L1 キャッシュを使用しないため、この問題は浮動小数点データのロード時に最も発生しやすくなる。

L2 バンク競合が発生すると、いくつかのアーキテクチャ・イベント・カウンタの値が加算される。これらのカウンタを観察すると、L2 バンク競合の発生を見つけ、その重大性を判断できる。L2_OZQ_Cancels1.Bank_CONF はバンク競合を伴う L2 アクセスの回数をカウントする。また、バンク競合があると両方のロードによって L2_Bypass の値が加算される。このため、バンク競合の発生はきわめて容易に発見できる。これらのカウント数と L2_Data_References.L2_Data_Reads を比較することで、L2 読み出しのうちバンク競合の発生した割合を把握できる。

VTune™ アナライザのドリルダウン機能を利用すれば、バンク競合を引き起こしているコード行を正確に特定できる。次に、デバッガ内でコードを実行すると、ロードされているデータのアドレスを分析できる。こうした情報をもとに浮動小数点データ・アクセスの順番を変えれば、バンク競合を防げる。

複雑なコードでは、VTune アナライザのイベント・スキッドによってバンク競合の詳細な位置を特定できないときがある。Bank_Conflict サブイベントの発生位置がイベント・スキッドによって移動していることが明らかな場合は、他のイベントを使用すればバンク競合の発生している場所を正確に特定できる。アドレスの競合およびそれに関連するバンク競合が起こるのは、1 サイクルに複数のロードが実行された場合である。そこで、VTune アナライザでソース・コードとディスアセンブリ・コードを混在表示させると、バンク競合の位置を正確に特定できる。また、Data_EAR_Event を使ってもバンク競合の位置を知ることができる。レイテンシが最小値よりも大きく、umask =1、レイテンシ >= 8 サイクルの条件でサブイベントを選択すれば、レイテンシの大きいアクセスの場所を特定できる。L3_READS.DATA_READ.ALL イベントを併用した場合、通常よりも高い比率というのが条件となる。これはつまり、L2 にミスしないレイテンシの大きいロードを意味している。もちろん、L2 にミスしたロードはすべてこのレイテンシの条件を満たす。なお、EAR イベントは発生位置については正確に計測できるが、発生回数については不正確なので注意が必要である。EAR イベントではハードウェアが細かいイベントのサンプリングを行うが、これはプログラムがキャッシュ・アクセス・ハードウェアをどのように利用しているかによって大きく影響を受けるためである。

浮動小数点データと L2 バンクの競合

FP データ競合に関するマイクロベンチマーク

浮動小数点ロード時にバンク競合が発生した場合のペナルティを測定するため、レイテンシに関するマイクロベンチマークを一部修正し、256 バイト境界にアライメントされたバッファを 2 つ使用するようにした。そして、レジスタ r36 に格納された 2 番目のバッファのベース・アドレスを 16 バイト単位で徐々に移動する上位のループ内で以下のアセンブラを実行する。これによって、複数のアクセスどうしの相対的アライメントをバンク幅の倍数でステップできる。これらのバッファが両方とも 256 バイト境界にアライメントされているなら、外側のループの最初の実行時に競合が発生し、以後、相対アライメントの移動を行う外側のループを 16 回実行するごとに競合が発生する。これまで同様、メモリ・アクセスによるストールの影響を調べるため、比較対象としてロード命令を含まないコードを右側に示している。なお、1 番目のロードについては通常の単独アクセス時のレイテンシでデータが読み出されるため、ここで測定されたレイテンシはすべて 2 番目のロードによるものである。

これは、今後紹介する複数メモリ・アクセスによるレイテンシの計測結果すべてに当てはまる。

```
do_read_inner:
{ .mmi
add r17 = r16, r33;;
add r29 = r30, r33
and r16 = r17, r34;;
}
{ .mmi
ldfs f26 = [r15]
ldfs f28 = [r31]
and r30 = r29, r34;;
}
{ .mfi
add r15 = r16, r36
mov f27 = f28
nop.i 0
}
do_read_branch:
{ .mfb
add r31 = r30, r35
mov f25 = f26
br.cloop.dptk.few do_read_inner;;
}
```

```
do_calibrate_inner:
{ .mmi
add r17 = r16, r33;;
add r29 = r30, r33
and r16 = r17, r34;;
}
{ .mmi
nop.m 0
nop.m 0
and r30 = r29, r34;;
}
{ .mfi
add r15 = r16, r36
mov f27 = f28
nop.i 0
}
do_calibrate_branch:
{ .mfb
add r31 = r30, r35
mov f25 = f26
br.cloop.dptk.few do_calibrate_inner;;
}
```

FP データ競合のペナルティと補正

このマイクロベンチマークを上位のループから実行すると、バンク競合によってレイテンシが 6 サイクル増えていることがわかる。この 6 サイクルのレイテンシは、ベース・アドレスを 16 バイト（または 32、48、64、…、240 バイトまで）シフトさせると解消できる。先に述べたとおり、バンク構造の幅は全体で 256 バイトである。

アクセス・モード	レイテンシ
シングル・アクセス	6 サイクル
2 つの同時アクセス（バンク競合なし）	6 サイクル
2 つの同時アクセス（バンク競合あり）	12 サイクル

浮動小数点ロードでバンク競合が発生した場合の CPI への影響の上限値は、次の関係式で計算できる。この式で得られる値は、実際の影響を多めに見積もることになるが、参考としては十分に役立つ。

```
CPI(FP Bank Conflicts) = L2_OZQ_Cancels1.Bank_CONF*3/IA64IR
```

L2_OZQ_Cancels1.Bank_CONF はバンク競合を起こしたロードをすべてカウントするため、キャンセルされたロードと再発行されたロードを二重にカウントしていることになる。したがって、6 サイクルのペナルティを 2 で割る必要がある。もちろん、これはバンク競合に起因するペナルティの上限値である。例えば、分析対象のアルゴリズムがループ構造をしており、そのループをアンロールした場合（あるいはコンパイラがアンループを行った場合）、命令のスケジューリングによって最小レイテンシよりも多くのレイテンシを吸収することができる。しかしながら、バンク競合を解消すれば OzQ の動作が減るため、L2 アクセスのスループットを高める効果が得られる。

ループのアンロールとバンク競合

多くの場合、バンク競合は比較的簡単に解決できる。ここでもう一度、Fortran における倍精度の行列の乗算を考えてみる。

```
Do k=1,MAX
    Do j=1,MAX
              Do i=1,MAX
              a(i,k)=a(i,k) + b(i,j)*c(j,k)
              enddo
    enddo
enddo
```

パフォーマンスを改善するために、まず内側のループをアンロールする。

```
 Do k=1,MAX
    Do j=1,MAX
              Do i=1,MAX
              a(i,k)=a(i,k) + b(i,j)*c(j,k)
              a(I+1,k)=a(I+1,k) + b(I+1,j)*c(j,k)
              a(I+2,k)=a(I+2,k) + b(I+2,j)*c(j,k)
              a(I+3,k)=a(I+3,k) + b(I+3,j)*c(j,k)
              enddo
    enddo
enddo
```

このコーディングによってパフォーマンスは大幅に改善されるが、同一のサイクルでメモリ内の連続した要素をロードするためにバンク競合が多発する。これは、アンロールした行を以下のようにインターリーブすれば簡単に解決できる。

```
Do k=1,MAX
   Do j=1,MAX
            Do i=1,MAX
            a(i,k)=a(i,k) + b(i,j)*c(j,k)
            a(I+2,k)=a(I+2,k) + b(I+2,j)*c(j,k)
            a(I+1,k)=a(I+1,k) + b(I+1,j)*c(j,k)
            a(I+3,k)=a(I+3,k) + b(I+3,j)*c(j,k)
            enddo
   enddo
enddo
```

(同じバンク（メモリ内の隣接したアドレス）からのロードを 1 サイクル以上離せるため、バンク競合は解消される。単精度データ（4 バイト）の場合、バンク競合を解消するには (j に対して）より大規模なアンロールを行い、より複雑なインターリーブを行う必要がある。

実際に上記の例を計測してみると、インターリーブによってパフォーマンスは 10% 以上改善されていることがわかる。

整数データと L2 バンク競合

キャッシュ可能な整数データへのアクセスは、常に L1-D キャッシュを通して行われる。このため、アドレス競合の問題、およびその競合がコア・パイプラインのフローに与える影響を把握するのはやや複雑になる。整数データのアドレス競合を理解するため、キャッシュからのデータ・フローについてここでもう少し詳しく解説する。

整数データのロード時にアドレス競合が発生するのは、複数の整数データ・アクセスが L2 キャッシュに対して同時にデータを要求した場合のみである。基本的に、L1-D に対して複数のデータ・アクセスが発生しても、問題にはならない。L1-D にはロード・ポートとストア・ポートが 2 つずつある。ロード・ポートは完全なデュアル・ポートであり、2 つのロード・アドレスを L1-D から同時に、競合なしに読み出せる。

ストアは 8 バイト幅の 8 つのグループで L1-D データ・アレイにアクセスする。ストアの場合は競合を起こす可能性があるが、こうした競合によるパフォーマンスへの影響を抑えるために専用のハードウェアが用意されている。

次に、別々のキャッシュ・ラインに対して複数の整数ロード・ミスが発生した場合のレイテンシについて考える、この場合、L2 から複数のデータを読み出すのに必要な全体的なレイテンシは、キャッシュ間の帯域幅によって大きく影響される。L2 キャッシュからデータを読み込む際は、要求されたデータ以外にも 64 バイトの L1 キャッシュ・ライン分のデータ全体が転送される。このため、複数のアクセス（複数のキャッシュ・ライン・フィルを必要とする）は L2 バンク間でアドレス競合を起こす可能性に加え、複数のキャッシュ・ラインの転送によるボトルネックを生じることにもなる。事実、複数の整数アクセスにアドレス競合がなくても、L1 キャッシュ・ラインは L2 から 1 度に 1 本しか更新できないため、余分なレイテンシが必要になってしまう。

アドレス競合にはさらに複雑な面がある。それは、要求されたデータが同一の L2 バンクに存在してはならないだけでなく、そのデータに関連したキャッシュ・ライン全体の中でもオーバーラップしてもアドレス競合が発生する点である。このため、整数データのアドレス競合に関して言えば、L2 キャッシュの動作は 64 バイト幅のバンク 4 つで構成されているように見える。浮動小数点データの場合とはデータ経路が異なる（L2 から直接浮動小数点レジスタ・ファイルにロードされる）ため、ペナルティもまた異なる。

整数データのバンク競合に関するマイクロベンチマーク

L2 から整数データをロードする際のアドレス競合を調べるため、浮動小数点データ・アクセスに関するマイクロベンチマークを一部修正し、整数のロードおよび移動命令を使用したのが、以下のマイクロベンチマークである。

```
        { .mmi
        add r17 = r16, r33;;
        add r29 = r30, r33
        and r16 = r17, r34;;
        }
        { .mmi
        ld4 r26 = [r15]
        ld4 r28 = [r31]
        and r30 = r29, r34;;
        }
        { .mmi
        add r15 = r16, r36
        mov r27 = r28
        nop.i 0
        }
        do_read_branch:
        { .mib
        add r31 = r30, r35
        mov r25 = r26
        br.cloop.dptk.few
        do_read_inner;;
```

```
        { .mmi
        add r17 = r16, r33;;
        add r29 = r30, r33
        and r16 = r17, r34;;
        }
        { .mmi
        nop.m 0
        nop.m 0
        and r30 = r29, r34;;
        }
        { .mmi
        add r15 = r16, r36
        mov r27 = r28
        nop.i 0
        }
        do_calibrate_branch:
        { .mib
        add r31 = r30, r35
        mov r25 = r26
br.cloop.dptk.few
do_calibrate_inner;;
```

ここでも、r36 に格納された 2 番目のバッファのベース・アドレスを調整すると、アドレス競合を解決できる。先ほど同様、2 番目のバッファのベース・アドレスを 16 バイト（L2 バンクの幅）単位で調整するループからアセンブラ関数を呼び出すメイン・プログラムを用意し、そこでこれを実行する。

整数バンク競合のペナルティ

整数データと浮動小数点データでは、競合のパターンが異なる。整数データの場合、2 つのアドレスのキャッシュ・ラインが同一の L2 バンクにオーバーラップしていると、競合が発生する。これはすなわち 1/4 の確率で発生する。浮動小数点の場合は 1/16 の確率でしか競合は発生しない。

このマイクロベンチマークを実行してみると、同一の L1 キャッシュ・ラインに対応した 64 バイトの範囲内で 2 つのアドレスがオーバーラップしている場合に、レイテンシ（2 番目のロードによって発生する）が 11 サイクルに増えていることがわかる。2 つのアドレスが上記の定義によるオーバーラップを起こしていない場合、レイテンシは 7 サイクルである。なお、L2 の整数データに対するシングル・アクセスを行った場合のレイテンシは 5 サイクルであり、コンパイラがスケジューリングの前提とする L1 へのアクセス時のレイテンシは 1 サイクルである。

アクセス・モード	レイテンシ
L1 からのシングル・アクセス	1 サイクル
L2 からのシングル・アクセス	5 サイクル
L2 からの 2 つの同時アクセス（オーバーラップなし）	7 サイクル
L2 からの 2 つの同時アクセス（オーバーラップあり）	11 サイクル

整数データと浮動小数点データのバンク競合

理論上は整数データと浮動小数点データの間でアドレス競合が発生する可能性もあるが、実際にそのようなケースはきわめてまれである。これは、浮動小数点ユニットは浮動小数点レジスタのみを使用し、整数 ALU は汎用レジスタのみを使用するためである。したがって、このような競合が起こるのは、バンドル内で実行される計算に整数データと浮動小数点データが入り交じっている場合に限られる。整数データと浮動小数点データの両方を 1 つの式で使うような計算の場合は、これら 2 つのデータ型を同時に使用できるようにするため、データ変換（ハードウェアによるデータのリキャスティング）が行われる。このため、コンパイラは整数データと浮動小数点データのロードを同一のサイクルで実行するようなスケジューリングは行わないはずである。複数のロードが 1 サイクルでも前後して OzQ に到着すれば、バンク競合は発生しない。

整数データと浮動小数点データのバンク競合に関するマイクロベンチマーク

整数と浮動小数点のバンク競合に関するマイクロベンチマークは簡単に作成できる。浮動小数点と整数のロードを同時に発行し、その次のサイクルで整数と浮動小数点の移動命令を発行すればよい。なお、これまで同様、計測されているレイテンシは 2 番目のロードによるものである。1 番目のロードは通常どおり完了する。マイクロベンチマークを実行して測定したレイテンシの値を以下の表に示す。

アクセス・モード	レイテンシ
L2 から整数のみをロード	5 サイクル
L2 から浮動小数点のみをロード	6 サイクル
整数と FP のロード（バンクのオーバーラップなし）	7 サイクル（両方のデータが利用可能になるまでのレイテンシ）
整数と FP のロード（バンクのオーバーラップあり）	11 サイクル

これを見ると、ペナルティの値は先ほどの整数データのバンク競合の場合と同じであり、CPI に対する影響も先に示した表と同じである。また、64 バイトのキャッシュ・ラインが L1 に読み込まれる際、L2 キャッシュがあたかも 64 バイト幅のバンクが 4 つのように動作するのも先ほどと同様である。

L2 に対する整数アクセスがバンク競合を起こした際のペナルティと補正

整数演算を多用するコードにおいて、L2 に対して複数の整数データ・アクセスを行い、バンク競合が発生した場合の CPI に対する影響については、次の近似式が成り立つ。なお、ここでは以下のとおり浮動小数点アクセスのバンク競合については無視できるものと仮定している。これまで同様、この関係式も参考程度にとどめる。

```
CPI(L2 Integer Bank Conflicts)=
                    L2_OZQ_CANCELS1.BANK_CONF*5/IA64IR
```

ここでも、重複カウントを解消するため、バンク競合のペナルティを 2 で割る必要がある。L2_OZQ_Cancels1.Bank_Conflict を 2 で割れば、バンク競合の回数を正しくカウントできる。

先ほど、浮動小数点データ・アクセス時のバンク競合として 2 つのバッファ間でバンク競合を起こすようなループ・コードについて説明したが、そのときと同様、整数データ・アクセス時のバンク競合（4 サイクルのペナルティ）を解決するには、2 つ目のバッファのベース・アドレスに対してパディングを追加すればよい。先ほど同様、まず 256 バイト境界にアライメントされたバッファのアドレス・アライメントを宣言する。

```
Buf = (UINT64) malloc(sizeof(mybuffer));
buf = buf + 256 - ((UINT64)buf%256);
```

次に、片方のバッファについて調整を行う。具体的には、このバッファが L2 構造における隣の 64 バイト領域に入るように移動する。

しかしこれよりも一般的に問題となるのは、隣接して格納された整数をロードするときである。この場合は、アクセス競合を解消するよりも、L1-D キャッシュ・ミスを起こさないようにする方が効果的である。しかし、この問題がストール・サイクルの主な原因となっているにもかかわらず、L1-D キャッシュ・ミスをどうしても解消できない場合は、以下の 2 つの方法を試す必要がある。

- 隣接するデータに対する最初の 2 つの参照を分離し、1 つのロードによってデータが L1-D キャッシュに読み込まれるようにする。
- 2 つのデータを別々のキャッシュ・ラインに取り込むようにする。このためには、複数の構造体を使用するなど、データ・ストレージの構造を変更する必要がある。

OzQ 再循環と複数の L2 キャッシュ・ミス

L2 内に存在しない（もちろん L1-D にも存在しない）キャッシュ・ラインに対するアクセスが同時にいくつも発生すると、2 番目以降のアクセスは OzQ 内を再循環する。最初のアクセスは L2 キャッシュ・ミスを発生させた後、L3（あるいは必要に応じてシステム・バス）にアクセスしてデータを取得する。2 番目以降のアクセスは L2 内のキャッシュ・ラインが更新されるまでの間、再循環状態に入る。

このような再循環がレイテンシにどのような影響を与えるかを調べるため、ここではバンク競合のマイクロベンチマークを一部修正して、2 つのベース・アドレスが同じバッファをポイントするようにして、相対オフセットは固定とした。このオフセットは、相対的なバンク・アクセスへの依存を調べる役目を果たすループ内のプログラムで設定されている。Itanium® 2 プロセッサのキャッシュ構造では、セカンダリ・ミスの処理は複雑なプロセスとなる。したがって、マイクロベンチマークで得られた数値は実際のアプリケーションの実行結果とは大きく異なる場合があり、本書で紹介するデータをもとに実際のアプリケーションにおける影響を推定するにはきわめて多くの不確実性が伴う。これまで同様、以下に示した結果はセカンダリ・ミスがアプリケーションの実行効率を大幅に低下させているかどうかを調べる際の参考程度にとどめておく。

このマイクロベンチマークによって得られた全体的なレイテンシをまとめると、以下のとおりとなる。

- L2 で発生した整数キャッシュ・ラインのセカンダリ・ミスが L3 にヒットした場合、両方のデータを読み出すまでのレイテンシは 25 サイクル。
- L2 で発生した浮動小数点キャッシュ・ラインのセカンダリ・ミスが L3 にヒットし、異なるバンクにアクセスした場合、両方のデータを読み出すまでのレイテンシは 28 サイクル。
- L2 で発生した浮動小数点キャッシュ・ラインのセカンダリ・ミスが L3 にヒットし、同じバンクにアクセスした場合、両方のデータを読み出すまでのレイテンシは 26 サイクル。
- L2 で発生した整数または浮動小数点キャッシュ・ラインのセカンダリ・ミスがメイン・メモリにヒットした場合、両方のデータを読み出すまでのレイテンシは 232 サイクル。

上記のレイテンシは、マイクロベンチマークで 2 つのロードを同一サイクルで発行して計測したものである。これらのレイテンシを見て、実行効率に大きな影響を与えているかどうかを調べれば、セカンダリ・ミスの再循環のペナルティがわかる。L3_READS.DATA_READ イベントはプライマリ・ミスの回数をカウントしているため、キャッシュ・ミスのペナルティについては本章ですでに説明した式によって、すでに明らかになっている。したがって、そこからどれだけのペナルティが増えているかを推定する必要がある。

レイテンシの増分は、L3 からデータを読み出せる場合で 11 ～ 14 サイクル、メイン・メモリから読み出す場合で 22 サイクルである。このため、セカンダリ・ミスによって増えたペナルティは、平均で 17 サイクルであると推定できる。ただしこの値はあくまでも参考程度にするように注意する。

セカンダリ・ミスに関連する発生イベントは、L2_FORCE_RECIRC、およびそのサブイベントである。

- L2_FORCE_RECIRC.SAME_INDEX:1サイクルで同一のアソシアティブ・セットに対して複数のアクセスが発生したことによって再循環が起こった回数を計測する。
- L2_FORCE_RECIRC.L1W：直前のサイクルでミスしたキャッシュ・ラインへの参照によって再循環が起こった回数を計測する。
- L2_FORCE_RECIRC.OZQ_MISS：2 ～ 5　サイクル前にミスしたキャッシュ・ラインへの参照によって再循環が起こった回数を計測する。
- L2_FORCE_RECIRC.FILL_HIT：3 サイクルよりも前にミスしてまだ更新の完了していないキャッシュ・ラインに対する参照が行われて発生した再循環の回数を計測する。

セカンダリ・ミスによる再循環をすべてカウントするには、これら 4 つのイベントをすべて使用する必要がある。ただしその場合、二重カウントや SAME_INDEX サブイベントの定義などが原因で、キャッシュ・ラインのセカンダリ・ミスによる再循環の回数を多めに見積もってしまう危険性がある。例えば、2 つの参照の間隔が 3 ～ 5 サイクルの場合は、OZQ_MISS と FILL_HIT サブイベントの両方がカウントされる。すでに更新待ちの状態にある同一のキャッシュ・ラインに対して 1 サイクルで複数の参照が発生すると、SAME_INDEX サブイベントがカウントされる、他の 3 つのサブイベントのいずれか 1 つと重複してカウントすることになる。また、このサブイベントは同一アソシアティブ・セット内の異なるキャッシュ・ラインに対して複数のキャッシュ・ミスがあった場合についてもカウントされる。

また、複数のミスの再循環が L3 ヒットに関連するものであるか L3 ミスに関連するものであるかを正確に判断はできない。したがって、平均値である 17 サイクルの数値を用いれば、セカンダリ・ミスによって増えるペナルティを推測できる。以上のような理由により、L2 のセカンダリ・ミスに起因する BE_EXE_BUBBLE への影響は、次の関係式で表せる。

```
Contribution to BE_EXE_BUBBLE = (L2_Force_RECIRC.Same_Index +
                                 L2_Force_RECIRC.L1W +

                                 L2_Force_RECIRC.OZQ_MISS +
                                 L2_Force_RECIRC.FILL_HIT)*17
```

また、CPI に対する影響は、これを IA64IR で割ると求められる。

セカンダリ・ミスによって増えるレイテンシを解消する方法は、本章ですでに述べたとおりである。最も明快で効果的な方法とは、以下のいずれかの手段によってキャッシュ・ミスによるレイテンシを完全になくすことである。

- 最適化オプション /O3 をつけてプログラムをビルドして、プリフェッチを生成する。
- プリフェッチ組み込み関数を使用する。

このほかにも、セカンダリ・ミスは以下の方法で回避できる。

- 同一の構造体または配列にアクセスしているソース行どうしを十分に離し、最初のアクセスによるキャッシュ・ラインの更新が完了してから 2 回目以降のアクセスを行うようにする。データが L3 でヒットすると仮定した場合、これには約 10 サイクルを要する。
  注意 : データがメイン・メモリにある場合にはこの手法は利用できない。
- あるいは、構造体の配列（または連結リスト）を配列の構造体で置き換えて、これらの要素が同一のキャッシュ・ラインに格納されるのを避ける。

BE_L1D_FPU_Bubble：L1-D および FPU マイクロパイプラインに起因するストール

7

BE_L1D_FPU_Bubble イベントは、L1-D および FPU に関連したマイクロパイプラインが原因でコア・パイプラインの DET ステージに発生したストール・サイクルをカウントする。

これまで同様、まず最初にこのイベントが発生する最大の要因となっているパフォーマンス・ボトルネックを把握する。そして次に、そのような実行効率の低下を解消するために適切な最適化を施していく。このカウンタは、メモリ・アクセスによるストール・サイクルを計測する点では BE_EXE_Bubble イベントと同じであるが、メモリ・アクセスのストールが起こるアーキテクチャ上の原因が異なる。このイベントでカウントされるストールは、単にコンパイラがレイテンシを完全に吸収できなかっただけでなく、データ転送にある種の閉塞が発生することに原因がある。したがって、これらのストールを解消するには、これまでとは違ったアプローチが必要となる。

BE_L1D_FPU_Bubble のサブイベントの階層構造

以下の表に、BE_L1D_FPU_Bubble のサブイベントの一覧を示す。このうち、最も一般的な原因となるのはデータ・キャッシュ・ユニット（DCU）の再循環（L1D_DCURECIR サブイベント）である。

この表は階層的に構成されている。

- 最上位の ALL は L1D と FPU の 2 つの要素に大別される。
- L1D はさらにいくつものサブイベントに分類される。ただし、実際のアプリケーションで主に発生するのはそのうちの 5 つである。

ここで注意が必要なのは、サブイベントには優先度はつけられていないため、あるサイクルが複数のサブイベントで二重にカウントされる場合があることである。しかし、たいていは分割したサブイベントの合計をとれば、BE_L1D_FPU_Bubble イベントのカウント数とほぼ一致する。

表 7-1　BE_L1D_FPU_Bubble のサブイベント

拡張子	PMC.umask [19:16]	説明
ALL	b0000	L1D または FPU によってバックエンドがストールした場合
FPU	b0001	FPU によってバックエンドがストールした場合
L1D	b0010	L1D によってバックエンドがストールした場合。これには、L1 マイクロパイプラインで発生したすべてのストール（コア・パイプラインの DET ステージに相当する L1 マイクロパイプラインの L1D ステージで発生したすべてのストール）が含まれる。
L1D_FULLSTBUF	b0011	ストア・バッファがフルになったため、L1D によってバックエンドがストールした場合
L1D_DCURECIR	b0100	DCU 再循環が原因で、L1D によってバックエンドがストールした場合
L1D_HPW	b0101	ハードウェア・ページ・ウォーカが原因で、L1D によってバックエンドがストールした場合
---	b0110	(* カウントは未定義 *)
L1D_FILLCONF	b0111	リターン・フィルと競合するストアが原因で、L1D によってバックエンドがストールした場合
L1D_DCS	b1000	ストールを要求している DCS が原因で、L1D によってバックエンドがストールした場合
L1D_L2BPRESS	b1001	L2 バック・プレッシャが原因で、L1D によってバックエンドがストールした場合
L1D_TLB	b1010	L2DTLB から L1DTLB への転送が原因で、L1D によってバックエンドがストールした場合
L1D_LDCONF	b1011	アーキテクチャ上の並べ替えの競合が原因で、L1D によってバックエンドがストールした場合
L1D_LDCHK	b1100	ロード・チェックの並べ替えの競合が原因で、L1D によってバックエンドがストールした場合
L1D_NAT	b1101	再循環 NaT 生成が必要な L1D データ・リターンが原因で、L1D によってバックエンドがストールした場合
L1D_STBUFRECIR	b1110	再循環が必要なストア・バッファ・キャンセルが原因で、L1D によってバックエンドがストールした場合
L1D_NATCONF	b1111	unat に書き込まれていない st8.spill と ld8.fill の競合が原因で、L1D によってバックエンドがストールした場合

BE_L1D_FPU_Bubble のサブイベントのうち、コンパイラが生成したコードでストール・サイクルを引き起こすものはそれほど多くない。これらのサブイベントは、NAT ビット、アプリケーション・レジスタ、制御レジスタ、load.acq/st.rel（ロードの取得 / ストアの解放）メモリ・フェンス命令などに関連したアクセス競合をカウントするためである。高級言語で作成したコードをコンパイルした場合、これらの要因が大きく影響することはあまりない。この種のサブイベントとしては、L1D_DCS、L1D_LDCONF、L1D_LDCHK、L1D_NAT、L1D_NATCONF がある。したがって、これらのサブイベントについては解説を割愛する。ただし、今後コンパイラ・テクノロジの進歩によって、スペキュレーションをより積極的に行うようになると、これらサブイベントのカウント数が大きくなるのも考えられる。

これらのイベントがストール・サイクルに大きな影響を与えていると判明した場合は、担当のインテル社員に問い合わせるか、あるいは https://premier.intel.com/ のサポート・ページを利用してアドバイスを受けることをお勧めする。

BE_L1D_FPU_Bubble を構成する L1D と FPU

BE_L1D_FPU_Bubble カウンタは、まず L1D と FPU の 2 つに分類される。これらのコンポーネントは、それぞれ L1D、FPU マイクロパイプラインで発生するストール・サイクルを計測する。上述のとおり、サブイベントには優先度が設定されていないため、これらサブイベントのカウント数を合計しても BE_L1D_FPU_Bubble カウンタの値とは厳密には一致しない。しかし近似値ではあっても、主要なボトルネックの発見と理解には十分役立つ。

ストール・サイクルの大きな原因となるのは、FPU よりも L1D の方が可能性が高いが、コーディング上の問題点もすぐに特定できるため、修正は容易である。一般に、L1D によるストール・サイクルは CPI の値に大きな影響を及ぼす場合が多い。

FPU サブイベントが、実際のアプリケーションや SPEC ベンチマークの個々のテストではほとんどカウントされない。このため、本章では主に L1D サブイベントについて解説する。ただしすべての情報を網羅しておくため、FPU について簡単に説明を行った後、L1D を説明する。

FPU

FPU サブイベントは、FPU マイクロパイプラインが動作の正確性を維持するためにストールしたサイクルをカウントする。このようなストールが起こる原因は 2 つある。最も一般的なのは、ある浮動小数点命令の結果を次の浮動小数点命令が入力として使用する場合など、命令どうしが連鎖しているケースである。

```
Fma  f33=f34,f35,f36;;
Fma  f37=f38,f39,f33
```

2 番目の fma 命令の入力として f33 を使用しているため、2 番目の命令は 1 番目の命令がデータを完全に返すまでストールする。これを命令のスコアボード・ストールと呼ぶ。出力レジスタのタグは、データの書き込みが完了するまでは無効に設定される。この場合、ストール・サイクルは BE_EXE_Bubble.FRALL カウンタで計測され、コア・パイプラインの EXE ステージのストールとしてカウントされる。

もう 1 つ、あまり多くないケースではあるが、パイプラインの DET ステージで例外を検出するためにストールする場合もある。浮動小数点の例外検出には数サイクルを要するが、その間により多くの発行グループがリタイアされる可能性がある。そこでプロセッサ・ステートの整合性を維持するために、SIR（Safe Instruction Recognition）ハードウェアが浮動小数点データを検査し、浮動小数点命令が安全に完了するかどうかを高速に判断する。SIR が安全性を確認した場合は、コア・パイプラインは中断なしに動作を継続する。一方、SIR ハードウェアが例外発生の可能性があると判断した場合は、パイプラインがストールする。この場合、ストール・サイクルは FPU マイクロパイプラインのストール・カウンタ、BE_L1D_FPU_Bubble.FPU でカウントされる。

このサブイベントのカウント数が大きい場合は、FP_TRUE_SIRSTALL イベントおよび FP_FALSE_SIRSTALL イベントを調べる。これらのイベントを見れば、SIR がパイプラインをストールしているかどうかを確認できる。

これらのストールを解消するには、浮動小数点結果がデノーマル数とならないように、計算を正規化する必要がある。これには、単精度を倍精度にシフトする、flush to zero オプションを有効にして再コンパイルする、あるいは極端に小さい数や大きい数を避けるようにアルゴリズムを変更する、などの方法がある。

L1D

L1Dサブイベントでカウントされるストール・サイクルの内訳は、そのほとんどがL1D_DCURECIR、L1D_L2BPRESS、L1D_TLB、L1D_HPW、L1D_STBUFRECIR、L1D_FULLSTBUFF の 5 つで占められる。一般に、プログラムでは読み出しよりも書き込みの方が少なく、ストア・バッファがフルになることはあまりない。通常は、データのロード、特にロード時の競合が問題となる。ここでは、まず最も単純なコンポーネントである L1D_FILLSTBUFF から解説する。

L1D_FULLSTBUFF カウンタについては詳細な説明は不要であろう。短いサイクルの間に非常に多くの整数ストアが発行されると、L1D ストア・バッファがフルになる。これにより、コア・パイプラインはストア命令の処理に余裕ができるまでストールする。この問題を解決するには、メモリへの書き込みが集中しないようにソース行どうしの位置を変更すればよい。これは、どのマイクロアーキテクチャにも通用するコーディング手法である。

L1D マイクロパイプラインのストールを引き起こすデータ・アクセス

通常、L1D サブイベントの中で特にカウント数の大きいのは L1D_DCURECIR、L1D_TLB、L1D_HPW、L1D_L2BPRESS、L1D_STBUFRECIR である。L1D マイクロパイプラインのストールを引き起こす要因としては主に、DTLB ミス、L2 OzQ のフル状態、ロード / ストア・スケジューリングの競合が挙げられる。本章ではこれら 3 つの要因について解説する。これらカウンタどうしの関係を理解するため、まず L1D マイクロパイプラインの詳細について若干の解説を行う。

L1-D キャッシュは、L2 OzQ の方式では要求のキューイングを行わない。したがって、要求が L1-D によって満たされない場合は、満たされるまで再循環される。要求が再循環されている間、L1D マイクロパイプラインの同期をとるためにコア・パイプラインがストールすると、このストールを L1D_DCURECIR（データ・キャッシュ・ユニットの再循環）サブイベントがカウントする。ここでは、データ・キャッシュ・ユニット（DCU）の再循環および L1D マイクロパイプラインのストールがなぜ発生したかを理解する必要がある。これは、別の L1D サブイベントのカウント数を調べれば明らかになる。

整数データへのアクセスには、L1-D キャッシュおよび 2 階層の TLB が密接に関係する。最初に L1-D キャッシュと L1DTLB の相互作用を解説し、その次に 2 つの TLB どうしの相互作用について説明する。

L1D_TLB と L1D_HPW について理解するには、仮想ページ・アクセス・システムのアーキテクチャに関する前提知識が必要となる。第 3 章の後半で解説しているので、参照のこと。

DTLB ミスによる影響の測定

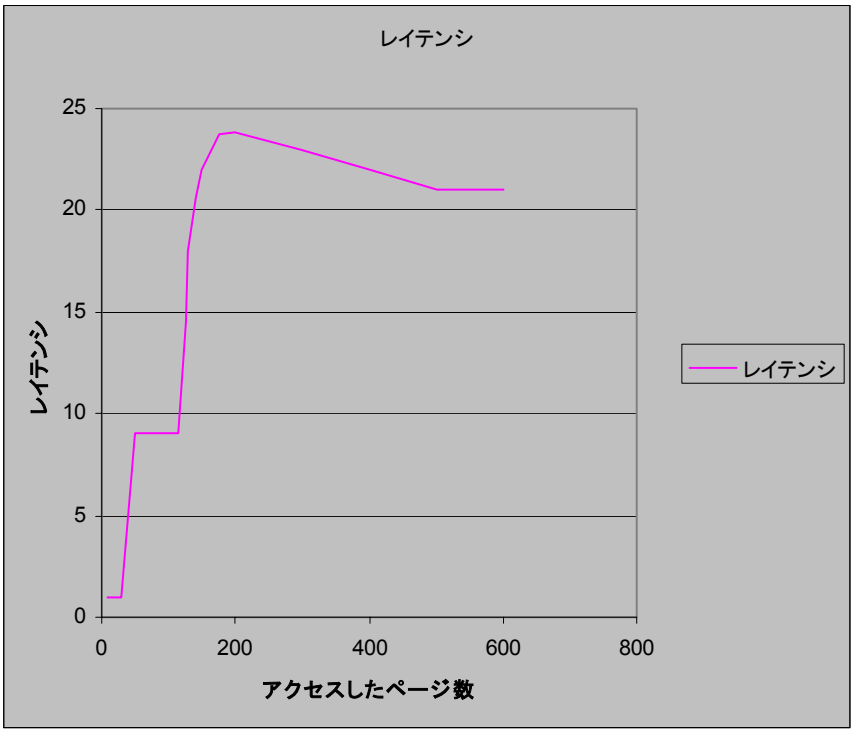
DTLB にミスして HPW が呼び出された場合の影響は、キャッシュ・アクセスのレイテンシを調べるために使用したマイクロベンチマークを一部修正すれば、容易にわかる。アドレスの増分幅を仮想ページのサイズ（ただし、すべてのアソシアティブ・セットを使用するために 1 キャッシュ・ライン分を加算）とすれば、毎回強制的に新しいページにアクセスできる。32 ページを超えるアクセスが行われると L1DTLB ミスが発生し、L1 キャッシュ・ミスが生じる。これは、L1 にヒットするためには有効な L1DTLB エントリが必要なためである。

必要なエントリが大容量の L2DTLB に格納されていれば、このエントリを L1DTLB に転送して L2 からキャッシュ・ラインの更新を行えばよい。この転送を 1 回行うごとに TLB カウンタの L1DTLB_TRANSFER が 1 回カウントされる。この場合、ストール・サイクルは全部で 9 サイクルとなる。このストール・サイクルのうち、一部は L1D でカウントされ（4 サイクル）、その内訳はサブイベント L1D_DCURECIR（3 サイクル）と L1D_TLB（1 サイクル）でカウントされる。残りの 5 サイクルは BE_EXE_Bubble.GRALL でカウントされる。これは、L2 にアクセスして L1 のキャッシュ・ラインを更新する際のサイクルである。

アクセスされるページ数が 128 に向かって増えていくと、L2DTLB にもミスが発生するようになる。これは、L2DTLB_MISSES イベント・カウンタで計測される。そしてこの時点でハードウェア・ページ・ウォーカ（HPW）呼び出され、データの検索を開始する。HPW は仮想ハッシュ・ページ・テーブル（VHPT）にアクセスするが、このとき VHPT エントリの 1 ページに対して L2DTLB エントリを 1 つ使用する。ほとんどのオペレーティング・システムは VHPT に 8 バイトのショート形式を使用しているため、標準的な 8KB のページ上には 1024 エントリが存在する。また、OS は最大でエントリの半分を確保できるが、実際に OS が確保するのはごくわずか（>5）である。しかしいずれにしても L2DTLB で利用可能なエントリの数は 128 ページよりも少なくなる。

このマイクロベンチマークを実行して、1 ページのサイズ（+1 キャッシュ・ライン）の増分幅で 200 ページにアクセスしたときのアクセス・レイテンシは 23 サイクルとなる。このうち 4 サイクルは BE_EXE_Bubble.GRALL でカウントされる。残りのストール・サイクルは BE_L1D_FPU_Bubble.L1D カウンタで計測される。さらにこのカウンタの内訳を見ると、そのほとんどは L1D_DCURECIR および L1D_HPW サブイベントで計測されているのがわかる。このような状況では、1 回ミスがあるごとに L2DTLB_Misses、DTLB_Inserts_HWP、DTLB_Inserts_HWP_Retired カウンタがすべて加算される。

次のグラフは、アクセスしたページ数に応じて整数ロード時のアクセス・レイテンシがどのように変化するかを示したものである。まず、L2DTLB が L1DTLB を更新するようになるとレイテンシが大きく増えている。次に、HPW が動作して仮想アドレスから物理アドレスへの変換を行うようになると、さらにレイテンシが増えていることがわかる。

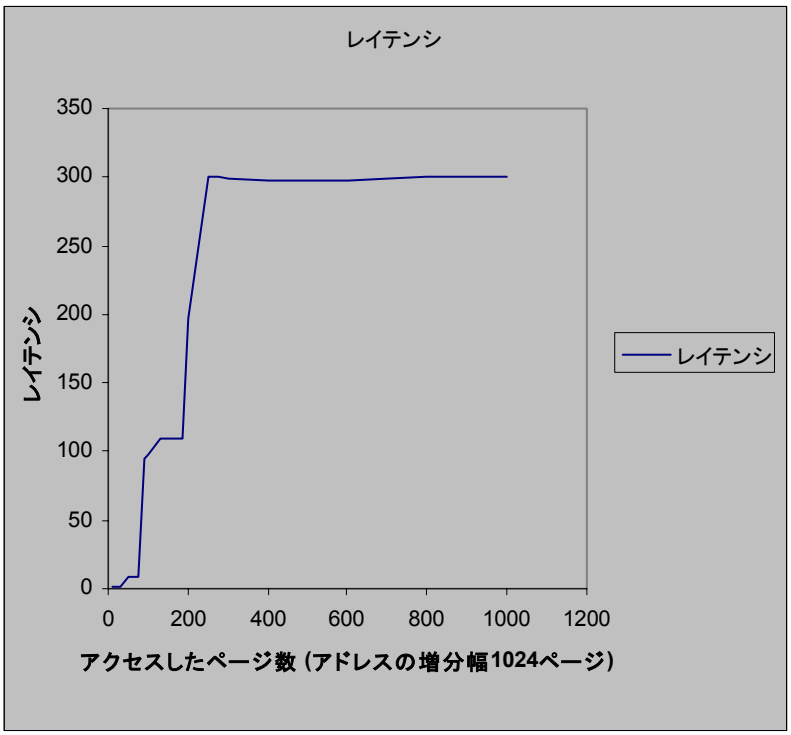


先ほどのマイクロベンチマークでは、アクセスするアドレスの増分幅を 1 ページ（+1 キャッシュ・ライン）としていたが、これを 1024 ページ（+1 キャッシュ・ライン）とすると、アクセス時のレイテンシは大きく異なってくる。9 サイクルのレイテンシで安定した状態の部分が見られるのは、L2DTLB が L1DTLB に対して必要な情報を転送し、L2 キャッシュがキャッシュ・ラインの更新を行える状態だからである。ページ数が 75 を超えると、レイテンシは上昇を始めるのは、HPW が L2DTLB 内に VHPT データを見つけられなくなったためである。

VHPT データのページを仮想アドレスから物理アドレスに変換するための情報が L2DTLB に見つからない場合は、VHPT フォルトが発生する。これは OS によって対処する必要があるが、このときのレイテンシは OS によって異なる。一般に、このハンドラはきわめて高速に動作する。以下に示すデータは、実際の計測値であり、単なる一例として見ていただきたい。

このグラフでは、例外ハンドラの 2 つのモードを観察できる。100 ～ 200 ページの部分で見られる最初の安定期では、データ取得のレイテンシが約 100 サイクルである。このレイテンシの多くは、例外の処理（17%）とフロントエンド・ストール（10%）が原因となっている。残りのレイテンシは BE_EXE_Bubble.GRALL と BE_L1D_FPU_Bubble に分割される。後者の大半は L1D_HPW サブイベントによるものである。この場合、L1D_HPW と L1D_DCURECIR サブイベントのカウント数はほぼ等しく、また、L1D 全体の合計にも等しいのは、2 つのサブイベントが同じサイクルをカウントしていることを意味する。

次に、ページ数が 200 を超えると、L3 ミスが大幅に増えてレイテンシは 300 サイクルにも達する。この増分のほとんどは BE_EXE_Bubble.GRALL（全体の 76%）で計測される。また、その他の 3 つの要素（L1D、FLUSH、FE）についても、それぞれ例外ハンドラの高速動作モード時に比べ 2 倍のカウント数となっている。



上記のいずれの場合も、スキャンするメモリ空間の範囲を小さくすると、実行効率を高められる。また、計算で使用するデータ・アクセスをブロック化してメモリ参照の局所性を高めるのも効果的である。L2DTLB ミスが発生するようになった場合は、ページ・サイズを大きくする方法もある。L1DTLB は 4KB 固定のページ・サイズしかサポートしておらず、システム・ページ・サイズがこれより大きくなると複数のエントリを使って対応している。L2DTLB と HPW/VHPT はこれよりもはるかに大きいページを扱えるため、ページ・サイズを大きくすればこられのコンポーネントがより広い空間のデータをカバーするようになる。これを行うには、Windows* .NET 用の OS API を使うか、あるいは Linux の場合はカーネルのリビルドを行う。

L1D_L2BPRESS サブイベント

L1D のサブイベントの中でおそらく最も複雑なのが、この L1D_L2BPRESS サブイベントである。L2 アクセス・キュー（OzQ）がフルになると、OzQ は L1D マイクロパイプラインに「バック・プレッシャ」を送り、それ以上のアクセス要求が L2 OzQ に送られないようにする。すると、OzQ に空きができてアクセス要求を受け付けられるようになるまで、L1D マイクロパイプラインはストールする。このストールによって、コア・パイプラインもストールする。このストール・サイクルをカウントするのが、L1D_L2BPRESS サブイベントである。このような理由でアプリケーションの実行効率が低下するのを防ぐには、プログラムの L2 アクセスの OzQ エントリがキューの中にとどまる時間を短くする必要がある。このためには、一般的に以下の事項について検討するとよい。

- キャッシュ・ミスを防ぐために適切なプリフェッチ命令が発行されているのを確認する。
- 明示的にループのアンロールや行の並べ替えを行い、バンク競合を防ぐ。
- L2 キャッシュに対するセカンダリ・ミスによる OzQ 再循環を減らす。
- プリフェッチ命令 / 組み込み関数には nt1 ヒントを使用して、エントリが OzQ 内にとどまる時間を短くする。

注 : 浮動小数点データや一時的なデータをプリフェッチする際など、L1D の使用が不適切な場合は、必ず nt1 ヒントを使うようにする。

コンパイラが生成するソフトウェア・パイプライニング・レポートおよび HLO レポートを見れば、これらの対策をコンパイラが自動的に行っている場所を把握できるため、コンパイラが自動的に行っていない場所に対して明示的に対策をとりやすくなる。したがって、これらのレポートが役に立つ場合が多い。

L1D_STBUFRECIR でカウントされるストールの原因

このカウンタは、ストアとロードの競合によるストール・サイクルを計測する。この種のストールは、ストア命令の 3 サイクル以内にロード命令が続き、両者が同じアドレスまたはキャッシュ・ラインを参照している場合に発生する。こうしたケースはあまり起こらないが、起こったとしても通常はソース行の並びを変えるだけで対処できる。

BE_Flush_Bubble：パイプライン・フラッシュによるストール

8

BE_Flush_Bubble カウンタは、パイプライン・フラッシュによって失われたサイクルをカウントする。この種のストール・サイクルの原因としては、次の 2 つがある。

- 分岐予測ミス
- 例外

ストールが分岐予測ミスによるものか例外によるものかは、それぞれ BE_Flush_Bubble.xpn、BE_Flush_Bubble.bru サブイベントによって直接調べられる。

以下の表に BE_Flush_Bubble カウンタのすべてのサブイベントを示す。

表 8-1　BE_Flush_Bubble のサブイベント

拡張子	PMC.umask	説明
ALL	bxx00	例外 / 割り込みまたは分岐予測ミスによるフラッシュが原因でバックエンドがストールした場合
BRU	bxx01	分岐予測ミスによるフラッシュが原因でバックエンドがストールした場合
XPN	bxx10	例外 / 割り込みによるフラッシュが原因でバックエンドがストールした場合
---	bxx11	(* 何もカウントされない *)

一般的に、パイプライン・フラッシュに起因するスループットの大幅な低下に対処するには、プロファイルに基づくフィードバックやプロシージャ間のインライン化を使用してコンパイルを行うのが最も効果的な方法である。これらは、第 9 章で詳しく解説する。この種のストール・サイクルの原因を調べるには、発生イベントを使用してパイプライン・フラッシュの詳細を明らかにしていく必要がある。

分岐予測ミスによるストール

BR_MISPRED_Detail カウンタを使用すれば、分岐予測ミスの詳細を調べられる。この親カウンタは、予測に成功したか失敗したかにかかわらず、すべての分岐をカウントする。詳細なデータを得るには、サブイベントごとに BR_MISPRED_Detail-BR_MISPRED_Detail2 を計算する。

```
( BR_MISPRED_Detail(umask=xxyy)-BR_MISPRED_Detail2(umask=xxyy) )
```

しかし実際にこれほど詳細な情報が必要とされるのは少なく、上記の減算によって得られるデータは無視できることが多い。そのような場合は、次式を使って分岐予測ミスの総数を求めるとよい。

```
BR_MISPRED_Detail(umask=0)-BR_MISPRED_Detail(umask=1)
```

ここで、umask = 0 はすべての分岐数、umask = 1 は予測に成功した分岐数を示している。なお、分岐予測ミスの正確な数値を得るには、BR_MISPRED_Detail2 による補正が必要となる。

```
(BR_MISPRED_Detail(0)-BR_MISPRED_Detail2(0))-
(BR_MISPRED_Detail(1)-BR_MISPRED_Detail2(1)
```

分岐予測ミスには数多くのサブイベントが存在する。これらを利用することで、予測を誤った分岐の方向やターゲットをより詳細に分析できる。以下の表に分岐予測ミスのサブイベントを示す。

表 8-2　分岐予測ミスのサブイベント

拡張子	PMC.umask	説明
ALL.ALL_PRED	b0000	予測結果に関わらず、すべての分岐タイプ
ALL.CORRECT_PRED	b0001	分岐予測に成功（結果およびターゲット）したすべての分岐タイプ
ALL.WRONG_PATH	b0010	予測した分岐方向の誤りが原因で分岐予測に失敗したすべての分岐タイプ
ALL.WRONG_TARGET	b0011	予測した分岐ターゲットの誤りが原因で分岐予測に失敗したすべての分岐タイプ
IPREL.ALL_PRED	b0100	予測結果に関わらず、すべての IP 相対分岐
IPREL.CORRECT_PRED	b0101	分岐予測に成功（結果およびターゲット）した IP 相対分岐
IPREL.WRONG_PATH	b0110	予測した分岐方向の誤りが原因で分岐予測に失敗した IP 相対分岐
IPREL.WRONG_TARGET	b0111	予測した分岐ターゲットの誤りが原因で分岐予測に失敗した IP 相対分岐
RETURN.ALL_PRED	b1000	予測結果に関わらず、すべてのリターン・タイプ分岐
RETURN.CORRECT_PRED	b1001	分岐予測に成功（結果およびターゲット）したリターン・タイプ分岐
RETURN.WRONG_PATH	b1010	予測した分岐方向の誤りが原因で分岐予測に失敗したリターン・タイプ分岐
RETURN.WRONG_TARGET	b1011	予測した分岐ターゲットの誤りが原因で分岐予測に失敗したリターン・タイプ分岐
NTRETIND.ALL_PRED	b1100	予測結果に関わらず、すべての非リターン間接分岐
NTRETIND.CORRECT_PRED	b1101	分岐予測に成功（結果およびターゲット）した非リターン間接分岐

表 8-2　分岐予測ミスのサブイベント

拡張子	PMC.umask	説明
NTRETIND.WRONG_PATH	b1110	予測した分岐方向の誤りが原因で分岐予測に失敗した非リターン間接分岐
NTRETIND.WRONG_TARGET	b1111	予測した分岐ターゲットの誤りが原因で分岐予測に失敗した非リターン間接分岐

分岐予測ミスに関するマイクロベンチマーク

以下のコードで示したような簡単なループを用いると、条件分岐の予測ミスによるペナルティを調べられる。

```
b1_2:
 {   .mfi
nop.m 0    nop.f 0
nop.i 0
 }
 {   .mfi
nop.m 0    nop.f 0
xor r11 = 1, r10;;
 }
 {   .mfi
nop.m 0
nop.f 0
nop.i 0
 }
 {   .mfi
nop.m 0
nop.f 0
cmp4.eq.unc
p7,p8=r11,r0;;
 }
 {   .mfi
nop.m 0    nop.f 0
nop.i 0
 }
 {   .mib
nop.m 0
nop.i 0
  (p8)
br.cond.sptk.clr
.b2_2
 }
```

```
.b1_2:
 {   .mfi
nop.m 0    nop.f 0
nop.i 0
 }
 {   .mfi
nop.m 0    nop.f 0
xor r11 = 1, r10;;
 }
 {   .mfi
nop.m 0
nop.f 0
nop.i 0
 }
 {   .mfi
nop.m 0
nop.f 0
cmp4.eq.unc
p7,p8=r11,r0;;
 }
 {   .mfi
nop.m 0    nop.f 0
nop.i 0
 }
 {   .mib
nop.m 0
nop.i 0
  (p8)
br.cond.sptk.clr
.b2_2
 }
```

```
.b2_2:
 {    .mfi
nop.m 0
nop.f 0
mov r10 = r11
 }
 {    .mfb
nop.m 0
nop.f 0
br.ctop.sptk .b1_2 ;;
 }
```

```
.b2_2:
 {    .mfi
nop.m 0
nop.f 0
nop.i 0
 }
 {    .mfb
nop.m 0
nop.f 0
br.ctop.sptk .b1_2 ;;
 }
```

レジスタ r10 はループに入る前に 0 に初期化される。左側のコードはループを実行するたびに分岐予測の成功と失敗を交互に繰り返す。右側のコードでは分岐予測ミスは発生しない。右のコードでは、左のコードにある mov r10=r11 命令を nop.i に置き換えれば、プレディケート p8 の値が反転しないようにしている。分岐しないと判断した場合は、経路はまったく同じであるものの、予測ミスとなってパイプラインがフラッシュする。ループを 1000 回実行するのに要するサイクルは、左側のコードで 7000 サイクル、右側のコードで 4000 サイクルとなる。左側のコードではループ実行のうち半分が分岐予測ミスであるため、プレディケート条件分岐の予測ミスは 6 サイクルのペナルティを生じているのがわかる。なお、実際の分岐予測ミスでは、上記のようなシンプルなループとは異なり、L1-I キャッシュ内に新しい命令を探しにいくためにペナルティがさらに大きくなる点に注意が必要である。

Itanium® 2 プロセッサでは必ず分岐予測ハードウェアが使用され、sptk 分岐ヒントは単に予測履歴テーブル（PHT）の初期化しか行わない。これは、Itanium プロセッサから変更になった点である。Itanium プロセッサでは、そのような分岐ヒントがある場合には分岐予測ハードウェアを使用しない。テーブルは、clr ヒントで毎回初期化される。

もちろん、コンパイラはこのような「悪い」コードは決して生成しない。

BE_RSE_Bubble：レジスタ・スタック・エンジンに起因するストール 9

レジスタ・スタックとして使用できる汎用レジスタは 96 個用意されている。しかし、コール・スタックが深い場合、あるいはレジスタを多用する関数によるコール・スタックなどの場合は、このリソースを使い切ってしまうことがある。このような場合には、レジスタ・スタック・エンジン（RSE）がレジスタに格納されている高レベルのコール・チェーンの値をバッキング・ストアにスピルして対処する。そして、コール・スタックがアンワインドされると、RSE がレジスタの値を復元する。以上の動作は RSE アクティビティと呼ばれ、必要に応じて自動的に行われる。このように RSE が呼び出されると、コア・パイプラインにストールが発生する。この種のストール・サイクルをカウントするのが BE_RSE_Bubble イベントである。なお、RSE がバッキング・ストアに退避するのは汎用レジスタのみである。FP レジスタについては、この方式でバッキング・ストアへのスワップは行われない。FP レジスタの場合は、生成されたコードによって明示的にスピルを行う必要がある。しかしこうしたケースはあまりない。FP レジスタが大量に必要とされるのは、過度にアンロールされたパイプライン・ループの実行など、ローカルな使用の場合に限られるためである。しかし、汎用レジスタは引数渡しのコール / リターン・メカニズムの土台として使われるため、コール・スタック・チェーンをアンワインドするためには復元可能にしておく必要がある。

BE_RSE_Bubble カウンタには多数のサブイベントがある。以下の表に、すべてのサブイベントの一覧を示す。ただし、RSE が呼び出された事実がわかるだけでも、対策は立てられることが多い。

表 9-1　BE_RSE_Bubble のサブイベント

拡張子	PMC.umask	説明
ALL	bx000	RSE によってバックエンドがストールした場合
BANK_SWITCH	bx001	バンク切り替えが原因で、RSE によってバックエンドがストールした場合
AR_DEP	bx010	AR の依存関係が原因で、RSE によってバックエンドがストールした場合
OVERFLOW	bx011	スピルが必要であるため、RSE によってバックエンドがストールした場合
UNDERFLOW	bx100	フィルが必要であるため、RSE によってバックエンドがストールした場合
LOADRS	bx101	loadrs 計算が原因で、RSE によってバックエンドがストールした場合
---	bx110- bx111	(* 何もカウントされない *)

Back_End_Bubble.ALL を構成する 5 つの要素の中で BE_RSE_Bubble が特に大きな割合を占めている場合は、ごく簡単なプログラミングの修正を行うだけでストールを解消できる。

RSE は、汎用レジスタの使用が多すぎると呼び出される。例えば、ごくわずかなルーチン、あるいは頻繁に使用されるカーネル・ルーチンに複雑な再帰アルゴリズムが含まれていると、汎用レジスタの使用量がきわめて多くなる。再帰チェーンは 1 つ 1 つのレベルで多くのレジスタを必要とするため、96 段のレジスタ・ファイルをすぐに使い切ってしまう。

以下の和を計算する簡単な再帰アルゴリズムを作成すると、RSE を強制的に呼び出せる。

```
1/2**n


double recursive(double x)
{
    double temp, epsilon=0.001
    temp=x/2.
    if(temp < epsilon) return 0.0
    return temp+recursive(temp)
}
```

上記の関数は自分自身を 10 回呼び出す。この関数を /Fa（Windows）オプションをつけてコンパイルして出力されるアセンブル・リストを編集する。具体的には、以下の alloc ステートメントで変更する。

```
alloc r33=ar.pfs,1,2,1,0
```

下記のように各レベルで 66 個のローカル・レジスタを割り当てる。

```
alloc r33=ar.pfs,1,65,1,0
```

BE_RSE_Bubble.All カウンタを使うと、各再帰レベルで RSE が呼び出されていることが確認できる。RSE の動作によってサイクル全体の約 1/3 を消費している。

RSE の動作を抑えるための方法

もう一度確認しておくと、RSE を呼び出す原因となっているのは、汎用レジスタの多用である。CPU_Cycles に占める BE_RSE_Bubble のカウント数の割合が高い場合は、以下のいずれかの対策をとる。

- 使用する汎用レジスタの数を抑えるようにアルゴリズムを簡略化する
- コンパイラのオプションを変更する

上記の例（きわめて意図的な例ではあるが）に示したとおり、プログラムの中でも特に CPU 負荷の高い部分に再帰アルゴリズムを使用すると、コール・スタックがきわめて深くなり、レジスタを解放する必要が生じる。どのようなアーキテクチャであっても、カーネルに再帰アルゴリズムを使用するのは避けるべきである。これは、深いコール・スタックをワインドしたりアンワインドするのは間違いなく効率の低下を招くためである。

もう 1 つ、RSE の動作を抑えるための手法として、複雑な計算の使用をやめ、その代わりに既に計算されたルックアップ・テーブルや補間を使用する方法もある。

RSE が呼び出されるもう 1 つの原因として考えられるのは、配列や変数を多用して、その 1 つ 1 つが固有のアドレスを必要とするような場合である。このような場合は、変数や配列をまとめて、必要なアドレスを少なくすると、問題を解決できる場合がある。あるいは、大規模で複雑な関数を、「リーフ・モジュール」と呼ばれる小規模でシンプルな関数に分割させても、レジスタの使用量を抑えられる。

IPO（プロシージャ間のインライン化）フラグの使用は、この問題を改善する場合もあるし、悪化させる場合もある。これはケース・バイ・ケースで判断する必要がある。モジュールをインライン化すると、重複は少なくなる。しかし一方で、呼び出し側の関数が複雑化して、コール・スタックの各レベルでさらに多くのレジスタが必要になるときもある。最善のソリューションは、試行錯誤によって見つける以外にない。プロシージャ間のインライン化は効果的であることが多いが、問題を引き起こすケースも確かに存在する。IPOf フラグ（Windows の場合は /Qipo）は、プロファイルに基づくフィードバック（/Qprof_gen、/Qprof_use）と組み合わせて使用した方が、よい結果が得られる。

このほか、パフォーマンスがそれほど重視されないルーチンの場合は、最適化レベルを下げる方法も有効である。最適化レベルを高めると、ソフトウェア・パイプライニングやループ・アンローリングを呼び出せるが、これらの最適化はいずれもアドレス保持のために汎用レジスタを多用する。したがって、モジュールがコール・スタックのきわめて高い場所に位置しており、しかも CPU サイクルをあまり必要としないのであれば、最適化レベルを下げて（/O2、/O1）コンパイルを行い、積極的な最適化（レジスタを多用する）を行わないようにすると、レジスタのリソースに対する負担を軽減できる場合がある。

Back_End_Bubble.FE：パイプラインのフロントエンドによるストール 10

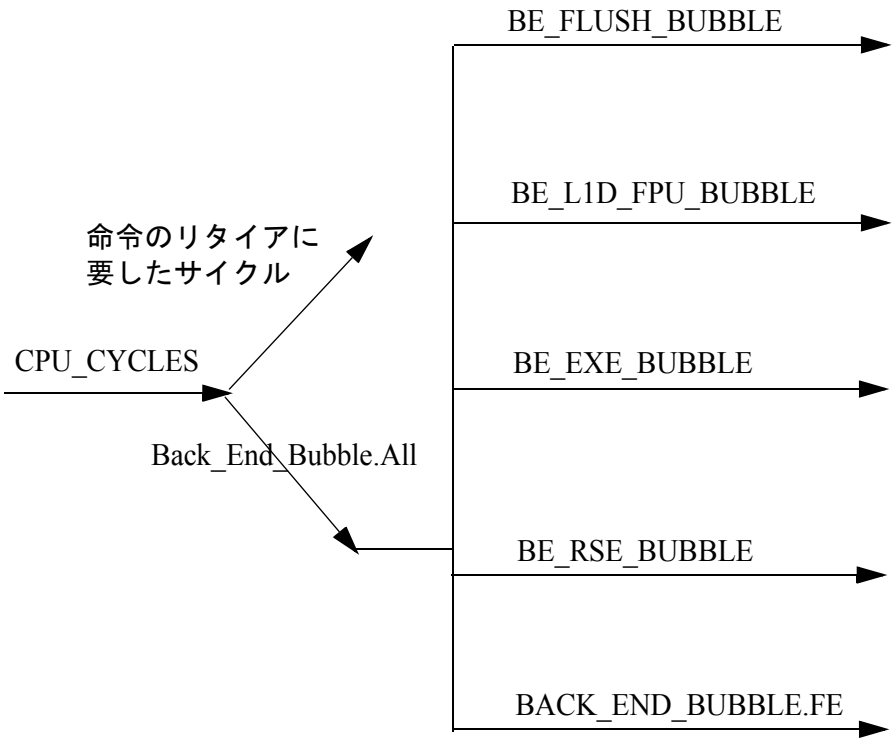
パイプラインのフロントエンドがバックエンドに対して命令を十分に供給できないために命令実行が停止し、ストール・サイクルが発生する場合がある。この種のストールは、Back_End_Bubble.FE カウンタで計測できる。通常、こうしたストールの原因はバイナリ・ファイルのレイアウトにある。バイナリ・ファイルのレイアウトが、アクティブなコード・ブロックを十分にグループ化できていない場合、命令キャッシュの使用効率が低下する。場合によっては、分岐予測ミスや例外がこの種のストールの原因となるときもある。Itanium® 2 プロセッサでは、常に分岐予測ハードウェアが使用され、分岐ヒントは単に予測履歴テーブルの初期化しか行わない。これは、Itanium プロセッサから変更になった点である。Itanium プロセッサでは、分岐ヒントを静的に使用する。したがって、Itanium 2 プロセッサでは Itanium プロセッサに比べ深刻な分岐予測ミスが発生することは少ない。

ソースを修正してこの問題を解決しようとするのは得策ではない。原理的には、「else if」ブロックのデフォルトのフォール・スルーをメインのフローにすれば一定の効果が得られる。しかしながら、プロファイルに基づくフィードバック（/Qprof_gen と /Qprof_use）およびプロシージャ間のインライン化（/Qipo）を行ってアプリケーションをリビルドすれば、ごくわずかな労力でかなりの成果が上がる。

付録

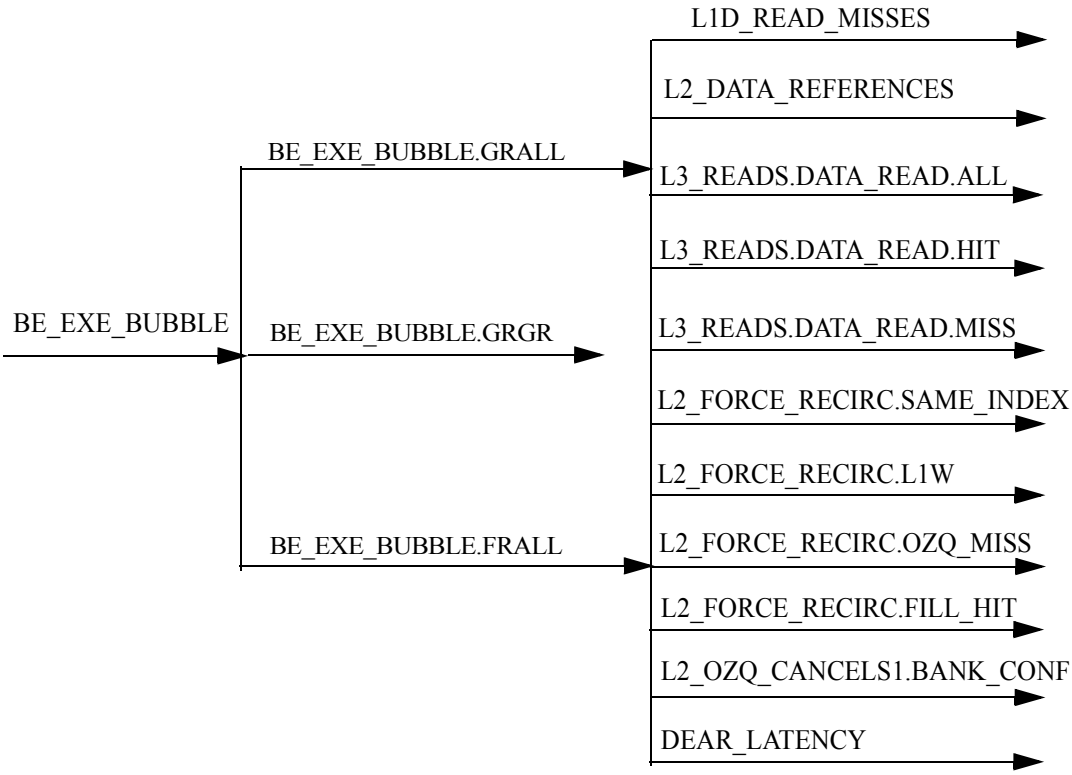
イベント・グループ

基本的なサイクル・アカウンティング



```
CPU_CYCLES
IA64_INST_RETIRED
BACK_END_BUBBLE.ALL
BACK_END_BUBBLE.FE
BE_FLUSH_BUBBLE
BE_EXE_BUBBLE
BE_L1D_FPU_BUBBLE
BE_RSE_BUBBLE
```

BE_EXE_BUBBLE の構成要素

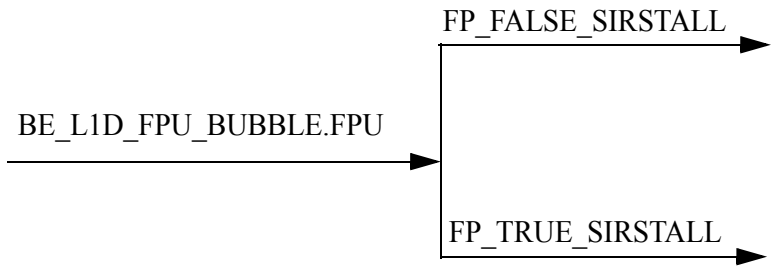



```
BE_EXE_BUBBLE
     BE_EXE_BUBBLE.GRALL
     BE_EXE_BUBBLE.GRGR
     BE_EXE_BUBBLE.FRALL
     L1D_READ_MISSES
     L1D_READS_SET0,1
     L2_MISSES
     L3_READS.DATA_READ.ALL
     L3_READS.DATA_READ.HIT
     L3_READS.DATA_READ.MISS
     L3_MISSES
     L2_REFERENCES
     L3_REFERENCES
     L2_OZQ_CANCELS0.ANY
     L2_OZQ_CANCELS1.BANK_CONF
     L2_FORCE_RECIRC.ANY
```

L2_FORCE_RECIRC.SAME_INDEX
L2_FORCE_RECIRC.L1W
L2_FORCE_RECIRC.OZQ_MISS
L2_FORCE_RECIRC.FILL_HIT
L2_FORCE_RECIRC.SNP_OR_L3
L2_GOT_RECIRC_OZQ_ACC
L2_ISSUED_RECIRC_OZQ_ACC
DEAR_LATENCY

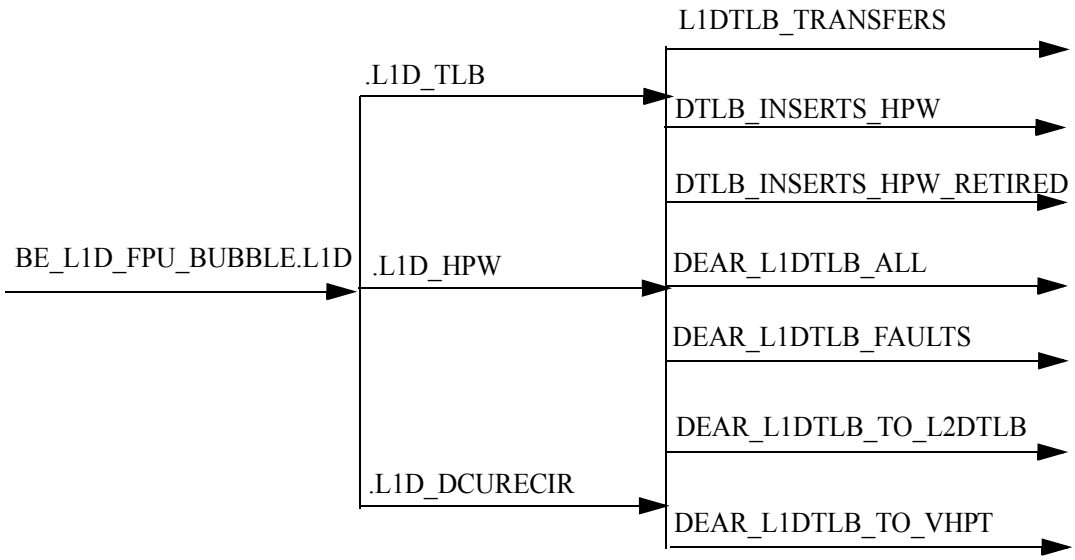
BE_L1D_FPU_BUBBLE

BE_L1D_FPU_BUBBLE.L1D の内訳



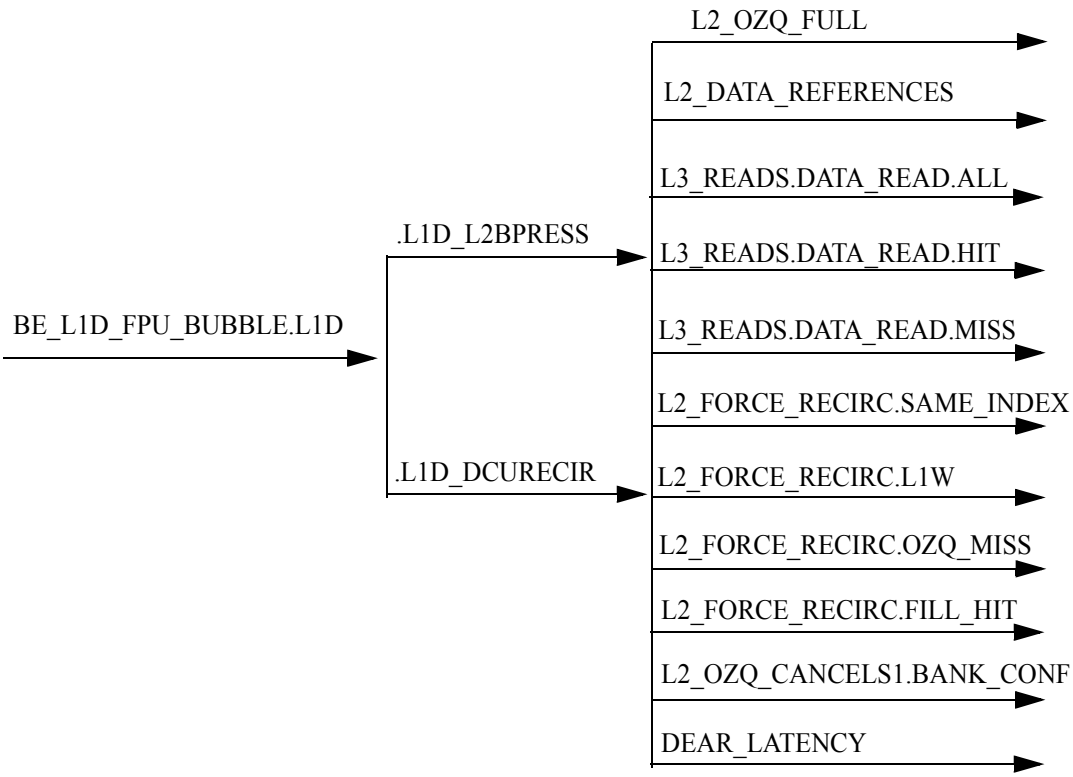
BE_L1D_FPU_BUBBLE.FPU
FP_FALSE_SIRSTALL
FP_TRUE_SIRSTALL

BE_L1D_FPU_BUBBLLE.L1D FROM DTLB の内訳



BE_L1D_FPU_BUBBLE.L1D
BE_L1D_FPU_BUBBLE.L1D_TLB
BE_L1D_FPU_BUBBLE.L1D_HPW
BE_L1D_FPU_BUBBLE.L1D_DCURECIR
L1DTLB_TRANSFER
L2DTLB_MISSES
DTLB_INSERTS_HPW
DTLB_INSERTS_HPW_RETIRED
DEAR_L1DTLB_ALL
DEAR_L1DTLB_FAULT
DEAR_L1DTLB_TO_L2DTLB
DEAR_L1DTLB_TO_VHPT

BE_L1D_FPU_BUBBLE.L1D_L2BPRESS の内訳



```
BE_L1D_FPU_BUBBLE.L1D_L2BPRESS
BE_L1D_FPU_BUBBLE.L1D_DCURECIR
L2_OZQ_FULL
L3_READS.DATA_READ.ALL
L3_READS.DATA_READ.HIT
L3_READS.DATA_READ.MISS
L3_MISSES
L2_REFERENCES
L3_REFERENCES
L2_OZQ_CANCELS0.ANY
L2_OZQ_CANCELS1.BANK_CONF
L2_FORCE_RECIRC.ANY
L2_FORCE_RECIRC.SAME_INDEX
L2_FORCE_RECIRC.L1W
L2_FORCE_RECIRC.OZQ_MISS
L2_FORCE_RECIRC.FILL_HIT
L2_FORCE_RECIRC.SNP_OR_L3
L2_GOT_RECIRC_OZQ_ACC
L2_ISSUED_RECIRC_OZQ_ACC
DEAR_LATENCY
```

BACK_END_BUBBLE.FE

BACK_END_BUBBLE.FE

FE_BUBBLE AND SUBEVENTS

FE_LOST_BW AND SUBEVENTS

IDEAL_BE_LOST_BW_DUE_TO_FE AND SUBEVENTS

BE_LOST_BW_DUE_TO_FE AND SUBEVENTS

BE_FLUSH_BUBBLE

ALL SUB EVENTS OF BE_FLUSH_BUBBLE

BRANCH_EVENT

BR_MISPRED_DETAIL AND ITS SUBEVENTS

BR_MISPRED_DETAIL2 AND ITS SUBEVENTS

BE_BR_MISPRED_DETAIL AND ITS SUBEVENTS

BE_RSE_BUBBLE

ALL SUB EVENTS OF BE_RSE_BUBBLE

RSE_EVENTS_RETIRED

RSE_REFERENCES_RETIRED AND ITS SUBEVENTS

用語集

C

clr ヒント
Clear ヒント。分岐履歴テーブルをクリアするために使われる。

D

DET ステージ
コア・パイプラインにおける検出ステージ。例外や分岐予測ミスの検出を行うほか、L1D や FPU マイクロパイプラインにおけるストールも検出する。

E

EAR イベント
Event Address Register イベントの略。命令ポインタやその他の情報をハードウェアでサンプリングを行う組み込みイベント。

EXE ステージ
命令を機能ユニットに対してディスパッチするコア・パイプライン・ステージ。

EXP ステージ
テンプレートを展開して機能ユニットを選択するコア・パイプライン・ステージ。

O

OzQ

L2 キャッシュへのアクセスを制御する 32 エントリのキュー。『インテル® Itanium® 2 プロセッサ・リファレンス・マニュアル：ソフトウェアの開発と最適化』(Revision 1.0) を参照。

P

PMC4

『インテル® Itanium® 2 プロセッサ・リファレンス・マニュアル：ソフトウェアの開発と最適化』(Revision 1.0) を参照。

PMD4

『インテル® Itanium® 2 プロセッサ・リファレンス・マニュアル ソフトウェアの開発と最適化』(Revision 1.0) を参照。

R

REN ステージ

alloc 命令、あるいはソフトウェア・パイプライニングを行うためのレジスタ名のローテートなどによって、レジスタのリネームを行うコア・パイプライン・ステージ。

RSE アクティビティ

レジスタ・スタック・エンジンが動作すること。

S

sptk 分岐ヒント

分岐すると静的に予測した (Statically predicted taken) 分岐ヒント

U

umask

ユーザ・マスク。イベントをさらに細かく選択するために使用するビット・パターン。

W

WRB ステージ

パイプラインのライトバック・ステージ。レジスタに格納された結果をデータ・キャッシュに書き戻す。

あ

アソシアティビティ

アソシアティブ・セットを構成するキャッシュ・ラインの数。

アドレス競合

データのアドレスが競合すること。アクセス時のレイテンシが通常よりも大きくなる。

か

仮想ハッシュ・テーブル (VHT)

オペレーティング・システムによって作成されるハッシュ暗号化されたテーブルで、これを用いてアプリケーションの仮想アドレスを物理アドレスに変換し、チップセットやディスク・システムを通じて物理メモリにアクセスできるようにする。

こ

コア・パイプライン
命令フローを制御する、バイナリの要求に応じて機能ユニットを割り当てる、データと命令を機能ユニットに同時に送り込む、などの処理を行う。

コール・スタック・チェーン
連鎖した関数呼び出しによって生まれる分岐の連続状態。

さ

サイクル・レイテンシ
レイテンシをサイクル数で表したもの。

サイクルのカウント
CPU サイクルをカウントすること。

す

ストア・ポート
データのストア時にキャッシュにアクセスするためのポート。

そ

総和則
いくつかの要素を合計すると必ず一定の数量になること。

は

ハードウェア・ページ・ウォーカ (HPW)

仮想ハッシュ・ページ・テーブル (VHPT) に格納されている仮想アドレスから物理アドレスへの変換データを参照する、プロセッサ・アーキテクチャの一部。

バイパス

通常の経路を短縮した高速データ・パス。

パフォーマンス・モニタ

特定のアーキテクチャ状態の発生をカウントするプロセッサ内部の監視用ハードウェア。『インテル® Itanium® 2 プロセッサ・リファレンス・マニュアル：ソフトウェアの開発と最適化』(Revision 1.0) を参照。

バンク

メモリの単位。通常は物理的なメモリの単位を指す。

ふ

分岐履歴テーブル

過去の分岐結果を格納し、分岐予測の参考に利用するテーブル。

ま

マイクロパイプライン

Itanium® 2 プロセッサには、3 階層のキャッシュからのデータを順番に並べたり、レイテンシの大きい浮動小数点およびマルチメディア命令を順番に処理するための専用マイクロパイプラインがそれぞれ用意されている。

め

命令テンプレート

128 ビットの命令バンドルに含まれる 5 ビットの情報。バンドルを構成する 3 つの 41 ビットの命令でそれぞれ必要な機能ユニットを指定する。また、バンドル内に命令グループの境界がある場合はその位置をストップ・ビット (;;) で記述する。

メモリ サブシステム

データを格納するコンピュータ・サブシステム。一般に、多階層のキャッシュ・メモリ、メイン・メモリ、ディスク・ドライブで構成される。

ゆ

ユニファイド・キャッシュ

命令とデータの両方を格納するキャッシュ。

り

リーフ・モジュール

他の関数を呼び出さない関数。

れ

レジスタ・スタック・エンジン

パイプラインの REN ステージの要求に応じて、汎用レジスタ・スタックのスピルとフィルをバッキング・ストアに対して行う。

索 引

インテル株式会社
〒300-2635 茨城県つくば市東光台 5-6
http://www.intel.co.jp/

2004 年 2 月

251464-001J