12 章
インテル® TSX の推奨事項

12.1 はじめに

インテル® トランザクショナル・シンクロナイゼーション・エクステンション（インテル® TSX）は、ロックベースのプログラミング・モデルを維持する一方、ロックで保護されたクリティカル・セクションのパフォーマンスを向上することを目的としている。

インテル® TSX を利用すると、プロセッサーはロックで保護されたクリティカル・セクションのスレッドをシリアル化する必要があるかどうかを動的に判断して、必要な場合にのみシリアル化を行うことができる。これにより、ハードウェアは、Lock Elision（ロックの無効化）として知られている手法を用いて、不要な動的同期によって損なわれているアプリケーションの並行性（コンカレンシー）を生かすことができる。

ロックの無効化により、ハードウェアは、開発者が指定したクリティカル・セクション（**トランザクション領域**とも呼ばれる）をトランザクション実行する。実行の際、ロックされた変数はトランザクション領域内で読み取られるだけで書き込まれない（したがって取得されない）。つまり、ロックされた変数はトランザクション領域で変更されないため、並行性を生かすことができる。

トランザクション実行に成功すると、トランザクション領域内で行われたすべてのメモリー操作はほかの論理プロセッサーからは瞬時に起こったように見える。プロセッサーは、コミットに成功した場合のみ、ほかの論理プロセッサーに見える領域内でアーキテクチャーの更新を行う。このプロセスは**アトミックコミット**と呼ばれる。トランザクション領域内で行われた変更は、アトミックコミットが行われることで、ほかの論理プロセッサーに見えるようになる。

成功したトランザクション実行ではアトミックコミットが保証されるため、プロセッサーは同期を行うことなく開発者が指定したコードセクションを安全だと推定して実行できる。特定の実行で同期が不要だった場合、スレッド間のシリアル化を行うことなく実行をコミットできる。

トランザクション実行が成功しなかった場合、プロセッサーは更新をアトミックにコミットできない。安全だと推定した実行に失敗すると、プロセッサーは実行をロールバックし、プロセスは**トランザクション・アボート**と見なされる。トランザクションがアボートすると、プロセッサーは領域で実行された更新をすべて破棄し、安全だと推定した実行が行われなかったかのようにアーキテクチャー・ステートを復元し、非トランザクションに実行を再開する。有効なポリシーに応じて、ロックの無効化が再試行されるか、処理を進めるため明示的にロックが取得される。

インテル® TSX には 2 つのソフトウェア・インターフェイスが用意されている。

- **Hardware Lock Elision** (HLE): 従来のプロセッサーと互換性のある命令セット拡張（XACQUIRE プリフィクスおよび XRELEASE プリフィクス）
- **Restricted Transactional Memory** (RTM): 新しい命令セット・インターフェイス（XBEGIN および XEND 命令）

従来のハードウェアでインテル® TSX 対応のソフトウェアを実行する場合は、HLE インターフェイスを使用してロックの無効化を実装する。一方、従来のハードウェアに対応する必要がなく、より複雑なロック・プリミティブを扱う場合は、インテル® TSX の RTM インターフェイスによりロックを無効化する。新しい命令を使用する後者のケースでは、トランザクション実行が行われない場合に備えて、開発者はトランザクション・アボートが生じた後に実行する（無効化されたロックを取得するコードを含む）非トランザクション・パスを常に提供する必要がある。

さらに、インテル® TSX には、論理プロセッサーがトランザクション実行しているかどうかをテストする XTEST 命令と、トランザクション領域をアボートする XABORT 命令も用意されている。

プロセッサーは、さまざまな理由によりトランザクション実行をアボートする。最も多い原因は、トランザクション実行している論理プロセッサーと別のプロセッサー間のデータアクセス競合によるもので、このようなアクセス競合はトランザクション実行の成功の妨げとなる。トランザクション領域内から読み取られたメモリーアドレスによりトランザクション領域の**読み取りセット**が構成され、トランザクション領域内に書き込まれたアドレスによりトランザクション領域の**書き込みセット**が構成される。インテル® TSX は、キャッシュライン単位で読み取りセットと書き込みセットを維持する。RTM を使用するロックの無効化では、明示的にロックを取得する別のスレッドがある場合でもトランザクション実行するスレッドが正しく動作するように保証するため、無効化されるロックのアドレスを読み取りセットに追加する必要がある。

別の論理プロセッサーがトランザクション領域の書き込みセットに含まれる場所で読み取りを行うか、トランザクション領域の読み取りセットまたは書き込みセットに含まれる場所で書き込みを行うと、データアクセス競合が発生する。これは**データ競合**と呼ばれる。インテル® TSX はキャッシュライン単位でデータ競合を検出するため、同じキャッシュラインに配置された無関係なデータ位置は競合として検出される。トランザクション・アボートは、トランザクション・リソースの制限により発生することもある。例えば、領域でアクセスされるデータの量が、実装固有の処理能力を超える場合などが挙げられる。CPUID や IO 命令などの一部の命令では、実装によりトランザクション実行が常にアボートされる。

インテル® TSX インターフェイスの詳細は、『Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1』(英語) の第 14 章を参照のこと。

以降の節では、インテル® TSX 命令を使用するソフトウェア開発者向けのガイドラインを説明する。ガイドラインでは、プリフィクス・ヒント(HLE) や新しい命令(RTM) を通じて、ロックで保護されたクリティカル・セクションの並行性を生かせるようにロックを無効化するインテル® TSX 命令の使用法に注目する。インテル® TSX 命令はロックの無効化以外にも利用できるが、それらの使用法はここでは説明しない。

以降、**ロックの無効化**という用語は、ロックを無効化する **HLE** ベースまたは **RTM** ベースの実装のいずれかを指す。

12.1.1 インテル® TSX の初期実装

インテル® TSX は、第 4 世代インテル® Core™ プロセッサーで初めて導入された。ここで示すガイドライン (読み取りセットと書き込みセットの入れ子と制御のハードウェア特性) は、インテル® マイクロアーキテクチャー Haswell† で導入されたインテル® TSX の初期実装に適用される。

第 4 世代インテル® Core™ プロセッサーは、L1 データキャッシュの**読み取りセット**アドレスと**書き込みセット**アドレスの両方を、キャッシュライン単位で追跡する。

読み取り/書き込みセットアドレスは、キャッシュの連想性の制限により、L1 データキャッシュからが追い出される (Eviction) ことがある。書き込みセットアドレスの追い出しはトランザクション・アボートにつながる。読み取りセットアドレスの追い出しは、対象のキャッシュラインが実装固有の L2 キャッシュで追跡されるため、常にトランザクション・アボートになるとは限らない。ただし、インテル® マイクロアーキテクチャー Haswell† ではバッファリングを保証していないため、ソフトウェアはバッファリングが保証されると仮定してはならない。

第 4 世代インテル® Core™ プロセッサーでは、L1 データキャッシュの連想性は 8 である。つまり、同じキャッシュセットへの 9 つ目の異なる場所に書き込むトランザクション実行はアボートする。しかし、マイクロアーキテクチャーの実装により、同じキャッシュセットへのアクセス数が 8 以下であっても、トランザクションがアボートしないという保証はない。

さらに、インテル® ハイパースレッディング・テクノロジーが有効な構成では、L1 キャッシュが同じコアの 2 つのスレッド間で共有されるため、同じコアの別の論理プロセッサーの操作によりキャシュラインが退避されることがある。

データ競合は、キャッシュ・コヒーレンス・プロトコルにより検出され、トランザクション・アボートにつながる。インテル® TSX の初期実装では、データ競合を検出したスレッドがトランザクション・アボートする。

HLE ベースのトランザクション実行でトランザクション・アボートが生じた場合、現在の実装では、ハードウェアは HLE 実行を開始した XACQUIRE プリフィクス命令を再開するが、XACQUIRE プリフィクスは無視する。その結果、ロックは無効化されずに再実行され、ロックは明示的に取得される。RTM ベースのトランザクション実行でトランザクション・アボートが生じた場合、現在の実装では、ハードウェアは XBEGIN 命令で示された命令アドレスから実行を再開する。

インテル® TSX の初期実装は入れ子を制限付きでサポートしており、RTM は 7 レベルまで、HLE は 1 レベルまでの入れ子がサポートされる。これは実装固有の値であり、同世代のプロセッサー・ファミリーの後の実装で変更される可能性がある。

『Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1』(英語) の第 14 章でも、トランザクション・アボートのさまざまな原因を詳細に説明している。インテル® TSX 命令およびプリフィクスの詳細は、『Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B』(英語) を参照のこと。

12.1.2 最適化の概要

ここからは、インテル® TSX 命令をロックの無効化に使用する際、マルチスレッド・アプリケーションを最適化およびチューニングするための推奨事項を示す。インテル® TSX は、ロックを取得した後のアプリケーションの動作を見落としがちなマイクロカーネルの代わりに、アプリケーションのパフォーマンスを向上させる (「12.2 アプリケーション・レベルのチューニングと最適化」を参照)。この後の節では、インテル® TSX を使用してロックの無効化を行う同期ライブラリーを作成する方法 (「12.3 インテル® TSX 対応の同期ライブラリーの開発」を参照)、インテル® TSX のパフォーマンス監視機能を効果的に使用する方法 (「12.4 インテル® TSX のパフォーマンス監視サポートを利用する」を参照)、そして初期実装のパフォーマンス・ガイドライン (「12.5 パフォーマンス・ガイドライン」を参照) についても述べる。

最初に、すべてのクリティカル・セクションのロックを無効化してから、問題のあるクリティカル・セクションを特定することを推奨する。この「ボトムアップ」アプローチによってアプリケーションの評価とチューニングが単純化され、開発者は適切なクリティカル・セクションに注目できる。

† 開発コード名

12.2 アプリケーション・レベルのチューニングと最適化

アプリケーションは通常、**同期ライブラリー**を利用してクリティカル・セクションに関連するロック取得とロック解放機能を実装している。これらのアプリケーションでインテル® TSX ベースのロックの無効化を活用する最も簡単な方法は、インテル® TSX 対応の同期ライブラリーを利用することである。インテル® TSX 命令は既存のライブラリーでも利用可能である。例えば、GNU* GLIBC pthread ライブラリーの新しいバージョン (GLIBC 2.17 以降) は、インテル® TSX 対応のミューテックス・ロックと reader-writer (読み取り/書き込み) ロックを使用するように変更されている。「12.3 インテル® TSX 対応の同期ライブラリーの開発」で、インテル® TSX 未対応のライブラリーをインテル® TSX 命令を使うように拡張する方法を示す。インテル® TSX 対応の同期ライブラリーは、従来の同期ライブラリーと同様に使用できる。

これらのライブラリーを使用するアプリケーションは、アプリケーションを変更することなくインテル® TSX を使用できるが、トランザクション実行のコミット率を増加させ、トランザクション・アボートによる無駄な実行サイクルを抑える基本的なチューニングとプロファイリングを行うことでパフォーマンスを向上できる。チューニングでは、最初にプロファイリング・ツール (「12.4 インテル® TSX のパフォーマンス監視サポートを利用する」を参照) を用いてアプリケーションのトランザクション動作の特性を把握することを推奨する。プロファイリング・ツールは、ハードウェアに実装されているパフォーマンス監視機能とサンプリング機能を利用して、アプリケーションのトランザクション動作に関する詳細な情報を提供する。パフォーマンス・モニタリング・カウンターやプリサイス・イベント・ベース・サンプリング (PEBS) メカニズムなどのプロセッサー機能が使用される。詳細は、『Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B』(英語) の第 18 章を参照のこと。

インテル® TSX 対応の同期ライブラリーを使用するアプリケーションは、従来の同期ライブラリーを使用する場合と同じ動作になる必要があるが、インテル® TSX によりレイテンシーが短縮されてスレッド間の同期が以前よりも速くなるため、コードの潜在的なバグが表面化する可能性がある。

12.2.1 最初のチェック

いくつかの簡単なチェックを行うことで、チューニングの労力を減らすことができる。特に、優れたライブラリーの利用とクリティカル・セクション内部の統計収集の扱いについては重要である。

- インテル® TSX 対応の優れた同期ライブラリーをアプリケーションで直接使用する。インテル® TSX 対応ライブラリーの上に独自のカスタム・ライブラリーを実装すると、トランザクション領域を識別できないことがある。インテル® TSX で同期ライブラリーを開発にする方法については、12.3 節を参照のこと。
- クリティカル・セクション内部の統計を収集しないようにする。クリティカル・セクション (および同期ライブラリー自身) は共有のグローバル統計カウンターを使用することがあるが、これらのカウンターはデータ競合やトランザクション・アボートを引き起こす。アプリケーションには通常、これらの統計収集を無効にするフラグが用意されているので、最初のチューニング段階でこれを無効にしておくと、本来のデータ競合に注目しやすくなる。

12.2.2 アプリケーションの実行とプロファイル

マルチスレッド・アプリケーションにおけるスレッドの同期に関連する相互作用を視覚化することは困難である。まず、インテル® TSX 対応の同期ライブラリーを使ってアプリケーションを実行し、パフォーマンスを測定する。次に、プロファイリング・ツールで結果を解析する。つまり、プロファイリング・ツールによりトランザクション実行されたアプリケーション・サイクルを測定し、アプリケーションのトランザクション実行の割合を把握するべきである (「12.4 インテル® TSX のパフォーマンス監視サポートを利用する」を参照)。

トランザクション実行の割合が低くなる要因はいくつかある。

- アプリケーションがクリティカル・セクションで同期をほとんど使用していない。この場合、ロックの無効化の利点が得られない。
- アプリケーションの同期ライブラリーがすべてのプリミティブにインテル® TSX を使用していない。クリティカル・セクションのロックの一部に内部カスタム関数やカスタム・ライブラリーを使用している場合、これらのロック実装をトランザクション実行に変更する必要がある (「12.4.2 無効化するロックを特定してすべてのロックが無効化されることを確認する」を参照)。
- アプリケーションが、ロックの無効化に対応した同期ライブラリーで提供されるものとは異なる、より高いレベルのロック構造 (本書ではメタロックと呼んでいる) を使用している。このような場合、その構造をロックの無効化に対応させる必要がある (「12.3.7 インテル® TSX を使用するアプリケーション固有のメタロックの無効化」を参照)。
- プログラムが LOCK プリフィクス命令をクリティカル・セクション以外で使用している。この場合、アルゴリズムがトランザクション実行に対応していない限り、インテル® TSX を利用できない。このようなロック目的以外での使用法については、本書では触れていない。

インテル® TSX パフォーマンス・チューニングの「ボトムアップ」アプローチでは、チューニング手法を次のようにモジュール化できる。

- ロックをすべて識別する。
- ロックをすべて無効化するインテル® TSX 同期ライブラリーを使用してプログラムを（変更せずに）実行する。
- プロファイリング・ツールでトランザクション実行を測定する。
- 必要に応じて、トランザクション・アボートの原因を調べて対処する。

12.2.3 トランザクション・アボートを最小限に抑える

プロファイリング・ツールのパフォーマンス監視機能を使用して、アボートしたトランザクション実行に費やされたサイクル数を計算する。すべてのトランザクション・アボートがパフォーマンスを低下させるとは限らない。別のスレッドが取得したロックを待機するために実行がストールしている場合や、トランザクション実行にデータ・プリフェッチ効果がある場合もある。

プロファイリング・ツールは、PEBS を用いてアボートしたトランザクション領域を認識し、相対的なコストに関する情報を提供する（「12.4 インテル® TSX のパフォーマンス監視サポートを利用する」を参照）。次に、トランザクション・アボートの一般的な原因と対策を示す。

チューニングの推奨事項 4: プロファイリング・ツールを用いて、パフォーマンス低下の多くの原因であるトランザクション・アボートを識別する。

トランザクション・アボートの原因には次のようなものがある。

- データアクセスの競合によるアボート
- ロック変数の競合によるアボート
- リソースバッファーの超過によるアボート
- HLE インターフェイス固有の制限によるアボート
- 『Intel® Architecture Instruction Set Extensions Programming Reference』（英語）の第 8 章で説明されているアボート。

12.2.3.1 データ競合によるトランザクション・アボート

別の論理プロセッサーがトランザクション領域の書き込みセットに含まれる場所で読み取りを行うか、トランザクション領域の読み取りセットまたは書き込みセットに含まれる場所で書き込みを行うと、データ競合が発生する。初期実装では、データ競合は、キャッシュライン単位で処理されるキャッシュ・コヒーレンス・プロトコルにより検出される。

ここでは、トランザクション・アボートの原因となるさまざまなデータ競合について説明する。これらのデータ競合は、回避可能なものもあるが、アプリケーションにもともと含まれるものもある。

フォルス・シェアリングによる競合

異なる変数を同じキャッシュライン（64 バイト）へ配置して複数のスレッドで個別に書き込みを行うと、フォルス・シェアリングが発生する。ハードウェアはキャッシュライン単位でデータ競合をチェックするため、アドレスがオーバーラップしていなくてもこれらの変数は同じアドレスを持つように見え、不要なトランザクション・アボートを引き起こす。

このフォルス・シェアリングの問題はインテル® TSX 固有の問題ではない。キャッシュ・コヒーレンス・プロトコルはシステム内でキャッシュラインを移動するが、これには大きなオーバーヘッドが伴う。少なくとも変数の 1 つが異なるスレッドによって頻繁に書き込まれる場合、異なる変数を同じキャッシュラインに配置するべきではない。

チューニングの推奨事項 5: 競合する変数はパディングを追加して別々のキャッシュラインに配置する。

チューニングの推奨事項 6: 可能な場合、フォルス・シェアリングが最小限になるようにデータ構造を再構成する。

実際の共有による競合

このトランザクション・アボートはフォルス・シェアリングによるものではなく、競合データが実際に共有されている場合に発生するが、ソフトウェアを変更することで軽減できることがある。次の節で、このような競合への対処方法を説明する。

統計による競合

一部のソフトウェアは、複数のスレッド間で共有されるグローバル統計カウンターを使用している。例えば、クリティカル・セクションのロックを取得したか、またはロックがすでに保持されていた回数をカウントしたり、複数のスレッドによってアクセスされるグローバル変数やオブジェクトの数をカウントしている。これらの統計はトランザクション・アボートの原因となる。このトランザクション・アボートに対処するには、統計の使用目的を理解する必要がある。

プログラムのロジックに影響しない場合は、統計を無効にしたり、条件付きでスキップすることができる。例えば、クリティカル・セクションのシリアル実行の回数をカウントするケースについて考えてみる。ロックが無効化されない場合、実行はすでにシリアル化されているため、クリティカル・セクション内で統計が更新される。ロックが無効化されている場合、ロックが無視された回数をカウントしても意味がない。この統計が役立つのは、ロックが無効化されなかった場合だけである。その場合、ソフトウェアは統計を利用してシリアル化のレベルを追跡できる。ロックを無効化しない実行の場合（つまり、シリアル化されている場合）のみ、XTEST 命令で統計を更新できる。これらの統計はプログラムの開発中にのみ有用であり、製品版では無効にできる。

しかし、場合によっては、統計を無効にしたりスキップできないこともある。これらの統計は、論理スレッドごとに維持することで、(フォルス・シェアリングを回避しつつ) 不要なトランザクション・アボートを回避することができる。このアプローチでは、統計の読み取り時にすべてのスレッドの結果をまとめる必要がある。これはまた、スレッド間の通信を最小限に抑えることで、インテル® TSX 命令を利用しなくてもアプリケーションのパフォーマンスを向上できる。

このほかにも、クリティカル・セクション外に統計を移動したり、アトミック操作で統計を更新する方法がある。これらのアプローチではトランザクション・アボートは軽減されるが、アトミック操作によりオーバーヘッドが発生したり、通信オーバーヘッドが軽減されない。

チューニングの推奨事項 7: グローバル統計は操作ごとに更新する代わりにサンプリングすることもできる。

チューニングの推奨事項 8: クリティカル・セクションでは不要な統計は避ける。

チューニングの推奨事項 9: クリティカル・セクションでは統計をスレッド単位で維持することを検討する。

プログラマーは、共有グローバル統計によるトランザクション・アボートを軽減するため最適なアプローチを選択する必要がある。初期テストでは、すべてのグローバル統計を無効にすると、グローバル統計が問題かどうかを判断しやすくなる。

データ構造内の資源管理による競合

データ構造内の資源管理による操作も、データ競合のよくある原因の 1 つである。例えば、データ構造は、構造内のエントリーの数を追跡するために変数を持つことがある。これは、統計カウンターと同様の影響があり、不要なトランザクション・アボートを引き起こす。

場合によっては（例えば、ヒープの再構成を引き起こすエントリーの数など）、アトミック更新を使用することでクリティカル・セクション外に更新を移動できる。

また、別のケースでは、データ競合が発生するタイミングを減らすアプローチを採用できることもある（「12.2.3.1 データ競合によるトランザクション・アボート」を参照）。

メモリー割り当ての競合

クリティカル・セクションでメモリー割り当てが行われることがある。その場合、スレッドのローカル領域にフリーリストを保持し、割り当て済みメモリーのフォルス・シェアリングを回避する、スレッド対応のメモリー・アロケーション・ライブラリーを使用することを推奨する。

条件付き書き込みによる競合の軽減

一般的なソフトウェアには、値がほとんど変わらない共有変数やフラグの更新が含まれる。そのような操作は、(値が同じ場合でも) キャッシュラインを更新するためキャッシュラインへの書き込み許可が必要になり、その共有変数へアクセスするほかのスレッドでトランザクション・アボートが発生する。このようなデータ競合は、値が同じ場合はストアを実行しないで、必要な場合のみ更新を実行するようにソフトウェアで回避できる（例 12-1 を参照）。

例 12-1 条件付き更新によるデータ競合の軽減

state = true; // 毎回更新する
var |= flag;
	if (state != true) state = true;
	if (!(var & flag)) var |= flag;

データ競合のタイミングの軽減

ここで示す手法では、頻発する実際のデータ競合によるトランザクション・アボートを回避できないケースがある。その場合、データ競合が発生するタイミングを短くすることを目標にすべきである。例えば、実際に競合するメモリーアクセスをクリティカル・セクションの最後に移動してタイミングを短縮できる。

12.2.3.2 トランザクション・リソースの制限によるトランザクション・アボート

インテル® TSX 実装は、共通のトランザクション領域の実行に必要なリソースを提供するが、トランザクション領域の実装の制限やデータ使用量によりトランザクション・アボートが生じることがある。アーキテクチャーがトランザクション実行に必要なリソースを保証したり、トランザクション実行の成功を保証することはない。

プロファイラーを利用して、処理能力の制限により頻繁にアボートするトランザクション領域を特定できる（「12.4.4 プロファイリング・ツールを利用してアボートを分類する」を参照）。ソフトウェアは、そのようなトランザクション領域内でデータの過剰なアクセスを避けるべきである。一般に、大量のデータアクセスには時間がかかるため、そのようなアボートは多くの実行サイクルを無駄にする。

クリティカル・セクションでのデータ使用量はアルゴリズムの変更により軽減できることもある。例えば、ソートされた配列には、リニア検索ではなくバイナリー検索を利用することで、クリティカル・セクション内でアクセスするアドレスの数を減らすことができる。

トランザクション領域の特定のコードパスで大量のデータにアクセスすることが想定される場合、アルゴリズムで（XABORT 命令により）早期にトランザクション・アボートを強制したり、無効化されたロックを取得してアボートせずに非トランザクション実行に遷移できる（「12.2.5 アボートが頻発するトランザクション領域またはトランザクション・パスへの対応」を参照）。

トランザクション領域内の動作が原因で、処理能力によるアボートが発生することがある。例えば、アプリケーションが初めてダイナミック・ライブラリー関数を呼び出す場合、ソフトウェア・システムはダイナミック・リンカーを呼び出してシンボルを解決する必要がある。初めての呼び出しがトランザクション領域内で行われる場合、大量のデータアクセスが生じ、通常アボートが発生する。このアボートは、ダイナミック・ライブラリー関数が初めて呼び出されるときにのみ発生する。これが頻繁に発生する場合は、非トランザクション実行中にトランザクション実行パスが使用されていないことが原因である可能性が高い。

12.2.3.3 ロック無効化固有のトランザクション・アボート

データ競合に加えて、ロック自体の競合によってトランザクション・アボートが生じることもある。その場合、クリティカル・セクションのトランザクション実行と非トランザクション実行がオーバーラップするタイミングを検出する必要がある。インテル® TSX を利用してロックを無効化する場合、ロックは読み取りセットに追加される。これは、HLE では自動で行われるが、RTM ではソフトウェア・ライブラリーで明示的に行う必要がある。これにより、明示的にロックを取得するほかのスレッドとの競合を確認できる。これは、アボートし、再開して、最終的にロックを取得するというトランザクション実行の自然な流れの一部である。

HLE と RTM を利用するロックの無効化では、このようにロック変数に対する二次的な競合が原因でアボートが発生することがよくある。トランザクション・スレッドのアボートは通常の非トランザクション実行に遷移し、その過程で明示的にロックを取得する。このロックの取得により、ほかのトランザクション実行スレッドをシリアル化しなければならなくなり、アボートする。

RTM の場合、ロックが解放されるのを待って取得を試みることで、フォールバック・ハンドラーはこれらの二次的なアボートを減らせる可能性がある（「12.3.5 RTM フォールバック・ハンドラーのガイドライン」を参照）。

12.2.3.4 HLE 固有のトランザクション・アボート

一部のトランザクション・アボートは、HLE ベースのロックの無効化でのみ発生する。この節では、このようなアボートについて説明する。

サポートされていないロックの無効化パターン

トランザクション実行を正常にコミットするには、ロックがある特性を持ち、ロックへのアクセスが特定のガイドラインに従っていなければならない。XRELEASE プリフィクス命令は、ロックの無効化の値を、対応する XACQUIRE プリフィクス命令（ロックの取得）が実行される前の値に復元する必要がある。これにより、ハードウェアは書き込みセットに追加することなく、安全にロックを無効化できる。XRELEASE プリフィクス命令（ロックの解放）と XACQUIRE プリフィクス命令（ロックの取得）のデータサイズとデータアドレスは、どちらも一致していなければならない。また、ロックはキャッシュ境界をまたぐことはできない。例えば、アドレス A への XACQUIRE プリフィクス命令（ロックの取得）の後に別のアドレス B への XRELEASE プリフィクス命令（ロックの解放）がある場合、アドレス A とアドレス B は一致しないためアボートする。

サポートされていない HLE 領域内のロック変数へのアクセス

通常、ロック変数は HLE 領域からアボートせずに読み取ることができる。しかし、ある種の一般的でないアクセスはトランザクション・アボートを引き起こすことがある。例えば、無効化されたロック変数へのアライメントされていないアクセスや一部がオーバーラップするアクセスは、トランザクション・アボートになる。この場合、無効化されたロック変数へ適切にアライメントされたアクセスが行われるように、ソフトウェアを変更すべきである。

HLE ベースのトランザクション領域内で無効化されたロックへ書き込みを行う場合は、必ず XRELEASE プリフィクス命令を使用する。そうしないと、トランザクション・アボートが発生する。

12.2.3.5 その他のトランザクション・アボート

プログラマーは、トランザクション領域内ですべての命令を安全に使用でき、すべての権限レベルでトランザクション領域を使用できる。ただし、一部の命令は常にトランザクション実行をアボートし、実行を非トランザクション・パスへ安全かつシームレスに遷移させる。そのようなトランザクション・アボートは、プロファイリング・ツールによって収集される PEBS レコードのトランザクション・アボート・ステータスには命令アボート (Instruction Aborts) として表示される (「12.4 インテル® TSX のパフォーマンス監視サポートを利用する」を参照)。

この命令の一覧は、『Intel® 64 and IA-32 Architectures Software Developer's Manual』(英語) に記載されている。一般的な例として、X87 およびインテル® MMX® 命令のアーキテクチャー・ステートに対する操作、セグメント、制御、デバッグレジスターを更新する操作、IO 命令、SYSENTER、SYSCALL、SYSEXIT、SYSRET などのリング遷移を引き起こす命令が挙げられる。

プログラマーは、トランザクション領域内では、X87/インテル® MMX® 命令の代わりに、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)/インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) 命令を使用すべきである。ただし、トランザクション領域内でインテル® SSE 命令とインテル® AVX 命令を混在させる場合は注意が必要である。XMM レジスターにアクセスするインテル® SSE 命令と YMM レジスターにアクセスするインテル® AVX 命令を混在させると、トランザクション領域のアボートの原因になる。VZEROUPPER 命令もアボートの原因になることがあるため、この命令はクリティカル・セクションの前に移動すべきである。

特定の 32 ビット呼び出し規約は、引数と戻り値の受け渡しに X87 ステートを使用している。プログラマーは別の呼び出し規約を利用するか、あるいは関数をインライン展開することを検討すべきである。long double 型などの一部のデータ型は X87 命令を使用するため、避けるべきである。

命令ベースによるアボートに加えて、ランタイムイベントによってトランザクション実行がアボートされることがある。

トランザクション実行中に非同期イベント (NMI、SMI、INTR、IPI、PMI、その他) が発生すると、トランザクション実行はアボートされ、非トランザクション実行に遷移する。そのようなアボートの発生率は、オペレーティング・システムのバックグラウンド・ステートに依存する。例えば、オペレーティング・システムのタイマーを用いて割り込みをかけると、トランザクション・アボートの原因になることがある。

トランザクション実行中に同期例外イベント (#BR、#PF、#DB、#BP/INT3 など) が発生すると、トランザクション実行はコミットされず、非トランザクション実行が必要になる。これらのイベントは発生しなかったかのように処理される。

ページフォルト (#PF) は通常、プログラムの起動時に発生することが多い。ページが初めて割り当てられるため、この間にトランザクション領域がアボートする確率が高くなる。このようなアボートは、プログラムが定常状態になると発生しなくなる。ただし、実行時間の非常に短いプログラムではこのアボートが頻発することがある。同様の現象は、大きなメモリー領域が割り当てられた際にも起こる。

トランザクション領域内でメモリーアクセスを行う場合、プロセッサーは参照するページ・テーブル・エントリーのアクセス (Accessed) フラグとダーティー (Dirty) フラグをセットしなければならないことがある。この処理は、ページへの初回アクセス時と書き込み時に起こる。現在の実装では、これらの処理はトランザクション・アボートを引き起こす。非トランザクション・モードで再実行すると、これらのビットが適切に更新され、通常、後続のトランザクション実行ではこれによるトランザクション・アボートが発生しない。このトランザクション・アボートは、PEBS レコードのトランザクション・アボート・ステータスには命令アボート (Instruction Aborts) として表示されるが、頻発しない限り特に注意する必要はない。

さらに、実装固有の条件やバックグラウンド・システム・アクティビティーがトランザクション・アボートの原因になることもある。例えば、システムのキャッシュ階層によるアボート、プロセッサーのマイクロアーキテクチャー実装との巧妙な相互作用、システムタイマーによる割り込みなどが挙げられる。このようなアボートは、インテル® TSX を使用するロックの無効化ではごくまれである。

チューニングの推奨事項 10: プログラム起動時のトランザクション領域では、定常状態よりも高い確率でアボートが発生する。

チューニングの推奨事項 11: バックグラウンド・アクティビティーにより、まれにオペレーティング・システムのサービスがトランザクション・アボートの原因になることがある。

12.2.4 トランザクション実行専用のコードパスの使用

RTM（フォールバック・ハンドラーを利用）や HLE と XTEST 命令を使用することで、プログラマーは非トランザクション・フォールバック・パスとは異なる、トランザクション領域でのみ実行されるコードを記述できる。

ただし、トランザクション実行時に実行されるコードが、非トランザクション実行時に実行されるコードと大きく異なる場合は注意が必要である。(命令およびデータの）ページフォルトなどのイベントや、アクセスビットとダーティービットを変更するページ操作は、トランザクション実行を繰り返しアボートする可能性がある。そのため、非トランザクション・フォールバック・パスでもこれらの操作が実行されるようにしなければならない。そうしないと、そのトランザクション領域はいつまでも成功できない。ロックの無効化により、アプリケーションのトランザクション・パスと非トランザクション・パスでは同じになり、唯一の違いは同期ライブラリーだけなので、これは一般的な問題ではない。

XTEST 命令は、トランザクション実行時にアボートの原因になる可能性の高い不要なコードシーケンスをスキップするのに利用できる。また、巻き戻された（unwind）コードや実際にロックを取得したときのみ必要なエラー処理コード（デッドロック検出など）をスキップして最適化を実装するのにも利用できる。

チューニングの推奨事項 12: トランザクション実行専用のコードパスは単純にしてインライン展開する。

チューニングの推奨事項 13: トランザクション実行専用のコードパスは最小限に抑える。

12.2.5 アボートが頻発するトランザクション領域またはトランザクション・パスへの対応

一部のトランザクション領域は高い確率でアボートし、これまでに示した手法では上手くアボートを減らすことができない。その場合は、この節に記載する手法を検討してみると良い。

12.2.5.1 アボートしないで非トランザクション実行へ遷移する

システムコールや IO 操作など、トランザクション・アボートが避けられないこともある。トランザクション実行パスでアボートが避けられない場合、ロックの無効化に RTM を利用しているソフトウェアでは、非トランザクション実行に遷移しロックの取得を試みて、成功した場合はトランザクション実行をコミットすることができる。例 12-2 に単純な例を示す。実際のコードでは、入れ子の処理など、追加の処理が必要になる。

例 12-2 アボートしないで非トランザクション実行へ遷移する

```
/* RTM トランザクションでトランザクション実行がアボートした場合は */
/* 非トランザクション実行でロックを取得する */

<オリジナルのロック取得コード>
_xend();  /* コミット */

/* アボート処理を行う */
```

12.2.5.2 早い段階によるアボートの強制

トランザクション領域内のアボートするパスにおいて、早い段階に PAUSE 命令または XABORT 命令を挿入し、強制的にトランザクション・アボートを発生させることで、破棄が必要な作業を最小限に抑えることができる。

12.2.5.3 選択したロックを無効化しない

ロックの無効化によりアプリケーションのパフォーマンスが低下し、どの手法でもトランザクション・アボートを軽減できない場合、トランザクション・アボートの発生率が高く、コストの高い特定のロックの無効化をソフトウェアでオフにできる。その場合は、アボートの発生率が高くてもパフォーマンスが向上することがあるため、アプリケーション・レベルのパフォーマンス指標を用いて検証すべきである。

12.3 インテル® TSX 対応の同期ライブラリーの開発

この節では、インテル® TSX 命令を使用してこれまでの同期ライブラリーをロックの無効化に対応させる方法を示す。

12.3.1 HLE プリフィクスの追加

XACQUIRE プリフィクスはクリティカル・セクションを保護するロックの取得に使用する命令の前に追加し、XRELEASE プリフィクスはクリティカル・セクションを保護するロックの解放に使用する命令の前に追加する。XRELEASE プリフィクス命令にはロックに対する書き込みも含まれている。この命令により、ロックの値が、対応する XACQUIRE プリフィクス命令でロックを取得する前の値に戻された場合、プロセッサーはロックの解放に関連付けられている外部書き込み要求を無視し、書き込みセットにロックのアドレスを追加しない。

12.3.2 無効化に適したクリティカル・セクションのロック

ライブラリー自体がデータ競合の原因にならないようにする必要がある。ライブラリーのデータ競合の一般的な例を次に挙げる。

- ロックの所有権フィールドの競合
- ロック関連の統計の競合

ロックの無効化に HLE を利用する場合、プログラマーは既存のコードパスにトランザクション実行を追加する必要がある (HLE では、ロックが無効化される場合もされない場合も、実行されるコードパスは同じであるため)。また、共有された場所への書き込み操作は、そのロック変数に対するロックの取得/解放命令によってのみ行われることを確認すべきである。共有された場所へのほかの書き込み操作は、通常、ロック無効化ライブラリーを利用して共通のロックを無効化する 2 つのスレッド間でデータ競合になる。これは、複数のスレッドを使って共有ロックで保護された空のクリティカル・セクションを繰り返し実行するテストによって簡単に特定できる。

12.3.3 ロックの無効化における HLE または RTM の使用

プロセッサーが HLE 拡張と RTM 拡張をサポートしているかどうかは、CPUID 情報によって判断できる。ただし、HLE プリフィクス (XACQUIRE と XRELEASE) は、プロセッサーが HLE をサポートしているかどうかを確認しなくても使用できる。HLE をサポートしていないプロセッサーはこれらのプリフィクスを無視し、トランザクション実行に入らずにコードを実行する。一方、RTM 命令 (XBEGIN、XEND、XABORT) は使用する前に、プロセッサーが RTM をサポートしているかどうかを確認する必要がある。RTM をサポートしていないプロセッサーで実行すると、これらの命令は #UD (未定義オペコード) 例外になる。XTEST 命令も、CPUID 情報をチェックして HLE か RTM のいずれかがサポートされていることを確認する必要がある。どちらもサポートされていない場合、この命令も #UD 例外になる。繰り返し確認しなくても済むように、CPUID 情報をある変数に格納しておくこともできる。

HLE では、トランザクション実行中のプロセッサーがクリティカル・セクションでロックの値を読み取ると、プロセッサーがロックを取得したように見える (トランザクション実行のものではない値が返される)。そのため、HLE 実行と HLE プリフィクスなしの実行の動作が同じになる。

RTM インターフェイスを利用して、プログラマーはより複雑な同期アルゴリズムを記述したり、トランザクション・アボート後の再試行ポリシーを制御することができる。RTM ベースのロック実装では、複数のコードパスを持つラッパーとして実装することが推奨される。つまり、1 つのパスで RTM ベースのロックを実行し、別のパスで RTM ベースでないロックを実行する (「12.3.4 ロック無効化に RTM を使用するラッパーの例」を参照)。通常、RTM ベースでないロックのコードを変更する必要ない。一度だけ再試行するプリミティブを使うことでパフォーマンスが向上する可能性がある。このプリミティブにより、スレッドはロックが解放された後にロックの無効化を再試行できる。

RTM 命令は明示的にロックに関連付けられていないため、ロックの無効化に RTM 命令を使用するソフトウェアは、トランザクション領域内でロックの状態を確認し、ロックがフリーの場合のみトランザクション実行を継続すべきである。さらに、ロックがフリーでないときの再試行ポリシーも定義する必要がある。

HLE との違いは、RTM ベースのクリティカル・セクション内でロックを読み取ると、ロックはフリーのままであり、取得されていないかのように見える。そのため、ロックの値を返すのに使用されるライブラリー関数は、トランザクション実行をアボートして非トランザクション実行の値を返す必要がある (「12.3.9 RTM ベースのライブラリーで無効化されたロックの値を読み取る」を参照)。HLE 命令では、明示的にロックアドレスが関連付けられており、正しい値が返されることがハードウェアによって保証されているため、このような状況は発生しない。

ユーザー/ソース・コーディング規則 37: ロックの無効化に RTM を利用する場合は、常にトランザクション領域内でロックをテストする。

チューニングの推奨事項 14: ラッパーでロック変数を読み取れない場合、RTM ラッパーは使用しない。

12.3.4 ロック無効化に RTM を使用するラッパーの例

この節では、RTM 命令を使用してロックを無効化するラッパーの記述方法を示す。既存のロック実装（無効化なし）をラッパーで囲み、ラッパー内に新しいパスを追加し、ロックを無効化することで、ラッパーでロックを無効化する場合としない場合のそれぞれのコードパスを記述する。ロックを無効化しないで取得するパスは、ロックを無効化するパスが成功しなかったときにのみ実行される。このアプローチでは、ロックを無効化しない場合のパスは変更されない。これは、チケットロックや reader-writer（読み取り/書き込み）ロックなどのさまざまなロックに適用できる。

例 12-3 にコードシーケンスの例を示す（使用されている組込み命令の説明は、「12.7 インテル® TSX 用の一般的な組込み関数 」を参照）。

例 12-3 ロックの取得/解放プリミティブに RTM を使用するラッパーの例

```
void rtm_wrapped_lock(lock) {
    if (_xbegin() == _XBEGIN_STARTED) {
        if (lock is free)
            /* 読み取りセットにロックを追加する */
            return; /* トランザクション実行 */
        _xabort(0xff);
        /* 0xff はロックがフリーでないことを示す */
    }
    /* トランザクション・アボートの後に実行されるコード */
    original_locking_code(lock);
}
```

```
void rtm_wrapped_unlock(lock) {
    /* ロックがフリーの場合は、ロックが無効化されたと仮定する */
    if (lock is free)
        _xend();    /* コミット */
    else
        original_unlocking_code(lock);
}
```

例 12-3 の _xabort() はロックがフリーでない場合、トランザクション実行を終了する。代わりに、_xend() を使用しても同様の動作になる。ただし、プロファイリング・ツールは _xabort() 操作とアボートコード 0xff（ソフトウェア規約）のほうを簡単に認識し、ロックがフリーでなかったケースであると判断できる。_xend() が使用されると、プロファイリング・ツールは、このケースとロックの無効化に成功したケースを区別することができない。

上記の例は、1 回だけ再試行し、トランザクション・アボートのさまざまな原因は区別しないという基本ポリシーを示すために単純化されている。より高度な実装では、トランザクション・アボートの原因に関する情報に基づいてロックごとに無効化を試すかどうかを決定するヒューリスティックを追加し、ロックがフリーでない場合、ブロックした後にロックの無効化を再試行するコードを追加する。これには、同期ライブラリーへのわずかな変更が必要になることがある。

場合によっては、プログラミング・エラーが原因で、スレッドが解放済みのロックを解放しようとすることがある。このエラーはすぐに明らかにならないことがある。しかし、前述のラッパーでロックの解放関数を RTM 対応のライブラリーに置換すると、XEND 命令がトランザクション領域外で実行され、#GP 例外になる。一般に、このエラーはオリジナルのアプリケーションで修正したほうが良いが、ソフトウェアでエラーになったコードパスを保持する場合は XTEST で XEND を保護できる。

12.3.5 RTM フォールバック・ハンドラーのガイドライン

RTM 用のフォールバック・ハンドラーは、RTM ベースのトランザクション実行が成功しなかったときに実行されるコードパスを記述する。インテル® TSX はトランザクション実行の成功を保証していないため、RTM フォールバック・ハンドラーは単にトランザクション実行の再試行を繰り返すのではなく、処理を進めることを保証しなければならない。

チューニングの推奨事項 15: ロックの無効化に RTM を利用する場合、ロックを取得することで処理が進むことを簡単に保証できる。

フォールバック・ハンドラーが明示的にロックを取得すると、そのロックを無効化する他のすべてのトランザクション実行スレッドがアボートし、実行がシリアル化される。これは、ロックがトランザクション領域の読み取りセットにあることを保証することで達成される。

EAX レジスターのアボート情報を参照して、トランザクション実行を再試行する場合とフォールバックして明示的にロックを取得する場合のヒューリスティックを作成できる。例えば、_XABORT_RETRY ビットが設定されていないと、トランザクション実行の再試行はアボートになる可能性が高い。フォールバック・ハンドラーは、このような場合とロックがフリーでない場合（例えば、_XABORT_EXPLICIT ビットが設定されているが、_XABORT_CODE()[1] が「ロックがビジー」状態であることを示す 0xff を返す場合）を区別し、ロックがフリーでない場合は待機後に再試行しなければならない。

アボートの原因がデータ競合（_XABORT_CONFLICT）であれば、ほとんどのデータ競合は一時的であるため、時間をおいて再試行することでパフォーマンスが向上する可能性がある。ただし、このような再試行の回数は制限し、無制限に繰り返すべきではない。

一般に、ロックがフリーでない場合、フォールバック・ハンドラーはロックがフリーになるまでトランザクション実行の再試行を待機すべきである。そうすることでロックを無効化しない非トランザクション実行にとどまるのを防ぐことができる。このような状況は、フォールバック・ハンドラーが、ロックがフリーでもトランザクション実行を試みることができないために生じる（「12.3.8 永続的な非トランザクション実行を回避する」を参照）。

ユーザー/ソース・コーディング規則 38: RTM アボートハンドラーは有効なテスト済みの非トランザクション・フォールバック・パスを提供する必要がある。

チューニングの推奨事項 16: ロックがビジー状態の場合は、ロックがフリーになるまで再試行を待機する。

12.3.6 インテル® TSX による無効化に適したロックの実装

この節では、一般的なロック・アルゴリズムにおいて、インテル® TSX 命令を使用するロックの無効化に適したバージョンを実装するアプローチについて述べる。この節で触れないアルゴリズムにも同様のアプローチを適用できる。

12.3.6.1 HLE を使用する単純なスピンロックの実装

スピンロックは、単純でよく使用されているロック・アルゴリズムである。このアルゴリズムでは、スレッドはロックがフリーかどうかを確認してから、LOCK プリフィクス命令でロックを取得する。ロックがフリーでない場合、スレッドはロックがフリーになるまで（通常は、ロックの値を保持するローカル・データ・キャッシュの読み取り操作を使って）スピンして待機する。

例えば、値がゼロのときはロックがフリーで、そうでない場合はほかのスレッドによって取得されていると仮定する。ロックは通常のストア命令によって解放される。

例 12-4 は、C11 規格に似た gcc* 4.8 以降の**アトミック組込み関数**を用いて、推奨されるアプローチに従ってスピンロックを実装している。このスピンロックで HLE を有効にするのに必要な変更は、__ATOMIC_HLE_ACQUIRE フラグと __ATOMIC_HLE_RELEASE フラグの追加だけで、残りのコードは HLE を利用しない場合と同じである。

[1] _XABORT_CODE は RTM アボートコードの xabort ステータスにアクセスする。

例 12-4 GCC* 4.8 以降で HLE を使用するスピンロックの例

```
#include <immintrin.h>  /* _mm_pause() に必要*/
/* ロックを 0 に初期化 */
void hle_spin_lock(int *lock)
{
    while (__atomic_exchange_n(lock, 1, __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE) != 0)
    {
    int val;
        /* 再試行する前にロックがフリーになるのを待機する */
        do {
            _mm_pause();  /* アボートのスペキュレーション */
            __atomic_load_n(lock,  &val, __ATOMIC_CONSUME);
        } while (val == 1);
    }
}
```

```
void hle_spin_unlock(int *lock)
{
    __atomic_clear(lock, __ATOMIC_RELEASE|__ATOMIC_HLE_RELEASE);
}
```

以下に、同じスピンロックを Windows* 用の C/C++ コンパイラー (Microsoft* Visual Studio* 2012 とインテル® C++ コンパイラー XE 13.0) の組込み関数を用いて実装する例を示す。

例 12-5 インテル® コンパイラーと Microsoft* コンパイラーの組込み関数で HLE を使用するスピンロックの例

```
#include <intrin.h>  /* _mm_pause() に必要 */
#include <imminitrin.h>  /* HLE 組込み関数に必要 */
/* ロックを 0 に初期化 */
void hle_spin_lock(int *lock)
{
     while (_InterlockedCompareExchange_HLEAcquire(&lock, 1, 0) != 0){
            /* 再試行する前にロックがフリーになるのを待機する */
            do {
                  _mm_pause();  /* アボートのスペキュレーション */
                  /* コンパイラーによる命令の並べ替えと待機ループのスキップを防ぎ
                     IA 上で追加のフェンス命令が生成されないようにする */
                  _ReadWriteBarrier();
            } while (lock == 1);
     }
}
```

```
void hle_spin_unlock(int *lock)
{
     _Store_HLERelease (lock, 0);
}
```

HLE スピンロックのアセンブラー実装については、「12.7 インテル® TSX 用の一般的な組込み関数 」を参照のこと。

12.3.6.2 インテル® TSX を使用する reader-writer (読み取り/書き込み) ロックの実装

reader-writer (読み取り/書き込み) ロックは、クリティカル・セクションの大部分が読み取り専用のときによく使用される。このロックはクリティカル・セクションの読み取りアクセスがシリアル化されるのを防ぐが、共有された場所に対してはアトミック操作 (LOCK プリフィクスが付加された XADD または CMPXCHG のことが多い) を要求し、複数の読み取り間で通信が必要になる。ロックの無効化は、読み取りと競合しない書き込みが通信することなく同時に処理できることを除いて、基本的にすべてのロックが reader-writer (読み取り/書き込み) ロックと同様の動作となる。

前述のラッパーアプローチにより、RTM を使って reader-writer (読み取り/書き込み) ロックを無効化することができる。ロックの無効化との唯一の違いは、reader-writer (読み取り/書き込み) ロックのアルゴリズムでは通常、読み取りと書き込みの両方の状態を確認してロックがフリーかどうかを判断していることである。そのため、読み取り/書き込みのそれぞれの状態を別々のキャッシュラインに配置できれば、読み取りのトランザクション実行と非トランザクション実行を並列に行える。読み取りは書き込みの状態が含まれないことを確認するだけで良い。

HLE を利用する場合、ロックが無効化されるときとされないときのコードパスは同じになる。一部の reader-writer (読み取り/書き込み) ロックの実装では実際のクリティカル・セクションではなく、読み取り/書き込み状態の保護にロックを使用している。この場合、まずロックを 1 つのアトミック操作から成る高速なパスに変更する必要がある。このパスでは、ロック変数が配置されているキャッシュラインは変更すべきではない。具体的には、読み取りの数と書き込みの数を 1 つのフィールドにまとめ、ロックの取得/解放関数に LOCK プリフィクスを付加した XADD 命令や CMPXCHG 命令を使ってこのフィールドをアトミックに確認/更新する。HLE プリフィクス (XACQUIRE と XRELEASE) を、LOCK プリフィクスが付加されたこれらの命令に追加する。興味深いことに、このアプローチは、インテル® TSX を使用しなくても reader-writer (読み取り/書き込み) ロックのパフォーマンスを向上させる。また、RTM ラッパーを使用することで、トランザクション実行用と非トランザクション実行用に異なるロックの取得パスが用意されるため、ロック構造の変更を回避できる。

チューニングの推奨事項 17: 読み取り/書き込みロックでは基本ブロックのロックではなく、ロック操作全体を無効化する。

12.3.6.3 インテル® TSX を使用するチケットロックの実装

チケットロックも一般的なアルゴリズムである。チケットロックはスピンロックのバリエーションで、共有する場所でスピンしてロックがフリーになったら取得する代わりに、チケットによってどのスレッドがクリティカル・セクションに入ることができるかを決定する。

前述のラッパーアプローチにより、RTM を使ってチケットロックを無効化することができる (「12.3.4 ロック無効化に RTM を使用するラッパーの例」を参照)。

一部のチケットロックの実装は、チケットの値が増えることを想定する。そのようなロックは、取得前と取得後のロックの値は同じでなければならないとする HLE 要件を満たさない。

チューニングの推奨事項 18: RTM を使用してチケットロックを無効化する。

12.3.6.4 インテル® TSX を使用するキューベースのロックの実装

一般に、ロックの無効化は複数のスレッドが共通のクリティカル・セクションへ同時に入り、コミットを試みることを前提としている。公正なロックでは、スレッドが先着順にクリティカル・セクションに入り、解放しなければならない。この 2 つの考え方は、場合によっては対立するように見えるが、一般的な目的はこれよりも柔軟性がある。

キューベースのロックはスレッドがロック要求のキューを作成する公正なロック方式であり、チケットロックの亜種とも入れる。

一部の実装では、最初の LOCK プリフィクス命令によりキューが作成される。その際、その命令に HLE XACQUIRE プリフィクスを追加してロックを無効化できる。トランザクション・アボートが発生しなかった場合、ロックの解放後にキューは空になる。しかし、トランザクション・アボートが発生し、アボートしたスレッドがロックを明示的に取得している場合 (つまり、キューが生成されている場合)、後続のスレッドはキューに追加され、ロックが解放されると先頭のスレッドのみがロックの無効化を試みる。さらに、別のスレッドがクリティカル・セクションに到達してキューに追加されると、トランザクション実行中のスレッドはアボートし、キューが空になるまで非トランザクション実行になる可能性がある。

これは、キューを使用してロックの無効化を試みる実装でのみ発生する。スピンフェーズを省略し最初のスピンが失敗した後にのみキューで待機する適応スピン・スリープ・ロックのような、最初のアトミック操作の後にのみキューを作成する実装には当てはまらない。この問題は、ラッパーを使用する実装 (RTM を使用するものなど) でも存在しない。これらの実装では、キューの処理でスレッドはロックの無効化を試みない。

チューニングの推奨事項 19: 最初のアトミック操作でキューを実装するロックに RTM ラッパーを使用する。

12.3.7 インテル® TSX を使用するアプリケーション固有のメタロックの無効化

一部のアプリケーションは、同期ライブラリーを利用してメタロックと呼ばれる独自のロックを構築する。アプリケーションは同期ライブラリーのロックによりメタロックのデータを保護し、データを更新したらロックを解放する。これは、「12.3.6.2 インテル® TSX を使用する reader-writer（読み取り/書き込み）ロックの実装」の reader-writer（読み取り/書き込み）ロックの実装のアプローチに似ている。

アプリケーションは、メタロックを保持しているときにクリティカル・セクションを実行し、メタロックを解放しているときは同期ライブラリーのロックを使ってメタロックデータを保護する。このシーケンスでは同期ライブラリーのロックを無効化しても意味がない。同期ライブラリー内のコードではなく、メタロック自体を無効化し、アプリケーションのコードをトランザクション実行すべきである。プロファイリング・ツールによってクリティカル・セクションを特定し、(「12.3.4 ロック無効化に RTM を使用するラッパーの例」に似た) RTM ラッパーによりロックの無効化中のメタロックを回避できる。

次のメタロックの実装例について考えてみる。

例 12-6 メタロックの例

```
void meta_lock(Metalock *metalock) {
    _lock(metalock->lock);
    /* メタロックを取得する */
       unlock(metalock->lock);
}
```

```
void meta_unlock(Metalock *metalock) {
  lock(metalock->lock);
 /* メタロックを解放する */
 unlock(metalock->lock);
}
```

```
  meta_lock(metalock);
 /* クリティカル・セクション */
  meta_unlock(metalock);
```

上記のコードは次のコードに置き換えられる。

例 12-7 RTM を使用するメタロックの例

```
void rtm_meta_lock(Metalock *metalock) {
        if (_xbegin() == _XBEGIN_STARTED)
                if (meta_state_is_all_free(metalock))
                    return;
                _xabort(0xff);
        }
        meta_lock(metalock);
}
void rtm_meta_unlock(Metalock *metalock) {
        if (meta_state_is_all_free(metalock))
            _xend();
        else
             meta_unlock(metalock);
}
```

```
rtm_meta_lock(metalock);
/* クリティカル・セクション */
rtm_meta_unlock(metalock);
```

チューニングの推奨事項 20: メタロックでは基本ブロックのロックではなく、外側のロックをすべて無効化する。

12.3.8 永続的な非トランザクション実行を回避する

トランザクション・アボートが起こると、最終的にロックを無効化しない非トランザクション実行に遷移する。これは、処理が進むことを保証している。ただし、特定の状況下と一部のロック取得アルゴリズムでは、スレッドがロックの無効化を試みずに非トランザクション実行にとどまることがあり、パフォーマンスの妨げとなる。

この状況を理解するため、HLE を使用する単純なスピンロックの実装例について考えてみる（同様のシナリオは RTM でも可能である)。ロックは値が 0 の場合はフリーで、値が 1 の場合は別のスレッドによって取得されている。

HLE を利用するロックの取得シーケンスは、次のように記述できる。

例 12-8 HLE を利用するロックの取得/解放シーケンス

```
  mov eax,$1
Retry:
  XACQUIRE;          xchg LockWord,eax
  cmp eax,$0               # 値が 0 ならロックの取得に成功
  jz Locked
SpinWait:
  cmp LockWord, $1
  jz SpinWait        # 値がまだ 1 のまま
  jmp Retry       # ロックがフリーなので取得を試みる
Locked:
```

```
  XRELEASE; mov    LockWord,$0
```

スレッドは、ロックを無効化できないときは無効化しないで取得する。別のスレッドが同じロックの取得を試みる場合、"XACQUIRE; xchg lockWord, eax" 命令を実行し、ロック操作を無効化して、トランザクション実行に入る。しかし、この時点でロックはほかのスレッドによってすでに取得されているため、このスレッドはトランザクション実行中に SpinWait ループに入る。これは、ハードウェアがクリティカル・セクション・ロックだと認識できず、ロック変数に対するアトミック操作と見なすためである。ハードウェアはロックがフリーでないということを知らない。

ロックを取得しているスレッドがロックを解放すると、そのロックへの書き込み操作によって、現在その場所をスピンしているスレッドがトランザクション・アボートになる（ロックの解放操作とトランザクション実行中のスレッドによるロックの読み取りループの間で競合が発生するため）。アボートすると、スレッドはロックを無効化しないで実行を再開する。これは、すべてのスレッドで起こる可能性がある。その場合、トランザクション実行中にスピンし、実際にロックがフリーになったら、ロックを無効化しないで非トランザクション実行する。そして、ほかにロックの取得を試みるスレッドがなくなるまで繰り返される。つまり、スレッドは非トランザクション実行にとどまることになる。

この問題への簡単な対処は、SpinWait ループで PAUSE 命令（アボートを引き起こす）を使用することである。これは、インテル® TSX を使用しない場合でもロックの解放の待機に推奨されるアプローチである。PAUSE 命令は SpinWait ループの非トランザクション実行を強制し、ロックが解放されたらスレッドがロックを無効化できるようにする。

例 12-9 HLE を使用する SpinWait の例

```
        mov eax,$1
Retry:
        XACQUIRE; xchg LockWord,eax
        cmp eax,$0 # 値が 0 ならロックの取得に成功
        jz Locked
SpinWait:
        pause      # pause 命令を挿入
cmp LockWord, $1
        jz SpinWait # 値がまだ 1 のまま
        jmp Retry   # ロックがフリーなので、取得を試みる
Locked:
```

チューニングの推奨事項 21: HLE スピンロックの待機ループには常に PAUSE 命令を使用する。

12.3.9 RTM ベースのライブラリーで無効化されたロックの値を読み取る

一部の同期ライブラリーは、ロックの値を読み取るインターフェイスを提供している。RTM を使用してロックを無効化するライブラリーは、ロックが読み取られるだけでライブラリー内へ書き込まれないため、無効化を行うスレッドがロック変数を取得したかどうかを正確に判断できないことがある。

場合によっては、ライブラリーのインターフェイスはロックが取得されているかどうかをチェックするだけの単純なもので、ソフトウェアに正当性チェックを提供する。RTM ベースのライブラリーを使用して関数に正しい値を確実に渡すには、トランザクション実行をアボートして明示的にロックを取得する必要がある。具体的には、XABORT 命令（_xabort(0xfe) を使用）でアボートを強制する。フォールバック・ハンドラーは 0xfe コードでこの状況を特定し、読み取りを排除する最適化を行える。また、_xtest() 組込み関数で不要なトランザクション・アボートを回避できる。

```
assert(is_locked(my_lock)) => assert(_xtest() || is_locked(my_lock))
```

無効化に対応した同期ライブラリー向けの効率良いプリミティブは、ロックの取得とロックの無効化を組み合わせられる。次に例を示す。

```
bool is_atomic(lock) { return _xtest() || is_locked(lock); }
```

また、動作を想定できる場合、ロック変数は関数の一部として読み取ることができる。例えば、ロックを取得する **try-lock** インターフェイスは、スレッドがロックの取得を 1 回だけ試みてロックがフリーかどうかを示す値を返す。これは、ロックを取得するためスピンを繰り返すスピンロックとは対照的である。一般に **try-lock** は問題にならないが、入れ子の try-lock により返される値に対しソフトウェアが暗黙の仮定を行っていることがある。RTM ベースの実装ではロックが無効化されるため、ロックがフリーであ

ることを示す値が返される。ソフトウェアでこの値について暗黙の仮定を行っている場合、同期ライブラリーは XABORT 命令で (_xabort(0xfd) を使用して) トランザクション・アボートを強制できる。ただし、これは一部のプログラムで不要なアボートを引き起こす。プログラムでこのような暗黙の仮定を行うことは推奨されない。また、このような暗黙の仮定が行われることはまれであるため、try-lock で同期ライブラリーによるアボートは推奨されない。

12.3.10 HLE と RTM を混在させる

HLE と RTM は、一般的なトランザクション実行機能に対する 2 つのソフトウェア・インターフェイスの選択肢を提供する。RTM 内で HLE を使用したり、HLE 内で RTM を使用する場合の動作は実装固有である。第 4 世代インテル® Core™ プロセッサーの初期実装では、HLE と RTM を混在させるとトランザクション・アボートにつながる。以降のプロセッサーでは動作が変わる可能性があるが、トランザクション・コミットのセマンティクスは維持される。

一般に HLE と RTM はロックを無効化するという目的は同じだが、ソフトウェア・インターフェイスが異なるため、アプリケーションでは混在しないようにすべきである。ロックを無効化するライブラリー関数は、呼び出し関数があるかどうか、また呼び出し関数が RTM あるいは HLE でロックを無効化しライブラリー関数を呼び出しているかどうかを把握していない可能性がある。

これらの状態は、_xtest() 関数によりソフトウェアで対応できる。例えば、ライブラリーがトランザクション領域内で呼び出されたかどうか、そしてロックがフリーかどうかを確認できる。トランザクション領域内で呼び出された場合は、新しいトランザクション領域を開始しないようにすることができる。ロックがフリーでない場合は、_xabort(0xff) 関数で結果を返せる。この場合、ロックの解放時に呼び出される関数は、取得操作がスキップされたことを認識できなければならない。

以下に、概念上のシーケンスを示す。

例 12-10 HLE と RTM を混在させる概念上の例

```
// ロックの取得シーケンス
// 関数またはスレッドのローカルを使用する
bool lock_in_transactional_region = false;
if (_xtest() && my lock is free) { /* すでにトランザクション領域内である */
  lock_in_transactional_region = true;
} else {
  // ロックがフリーの場合は取得し、そうでない場合はアボートする
}

// ロックの解放シーケンス
if (!lock_in_transactional_region) {
        // ロックを解放する
}
```

12.4 インテル® TSX のパフォーマンス監視サポートを利用する

インテル® TSX を使用してアプリケーションをチューニングする際、トランザクション実行動作とトランザクション・アボートの原因を理解するため、パフォーマンス・カウンターベースのプロファイリングを利用する。インテル® TSX で優れたパフォーマンスを得るには、プロファイリング・ツールで収集したデータを基にアボートを最小限に抑えるチューニングが必要になる。通常、アプリケーションのインストルメンテーションよりも、パフォーマンス・カウンターのほうが簡単でコード変更が少ないため推奨される。18.10.5 節と『Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B』(英語) の第 19 章に、現在の実装でサポートされる各種パフォーマンス監視イベントの情報が記載されている。

一般に、プロファイリング・ツールは情報を収集するために周期的に割り込みをかけるが、この割り込みによってトランザクション・アボートが発生するので、プロファイリングはトランザクション実行に影響する。そのため、プロファイリングではこの影響を最小限に抑えるべきである。ただし、トランザクション・アボートのみをプロファイリングする場合は問題にならない。

プログラムの起動時に一度しか実行されない多数のイベントがある。複雑なプログラムのプロファイリングでは起動時のプロファイリングをスキップすることで、これらのイベントによる不要なデータの収集を大幅に減らせる。

12.4.1 トランザクションの成功を測定する

最初に、アプリケーションのトランザクションの成功を測定する。これは、次の3 つのカウンターと Unhalted_Core_Cycles イベントを設定することで測定できる。

1. 固定サイクルカウンター (IA32_FIXED_CTR0) で FixedCyclesCounter を測定する。
2. IA32_PERFEVTSEL2 に IN_TX フィルターと IN_TXCP フィルターを設定して、IA32_PMC2 の CyclesInTxCP in を測定する。
3. MSR IA32_PERFEVTSELx (x= 0, 1, 3) に IN_TX フィルターを設定して、対応するカウンターの CyclesInTXOnly を測定する。

サンプリングはトランザクション・アボートを引き起こす可能性があるため、これらのサイクルの測定にはサンプリングではなくカウンターを使用する。この 3 つの値から、合計サイクル数、トランザクション実行で費やされたサイクル数、アボートされたトランザクション領域で費やされたサイクル数を計算できる。

```
CyclesTotal = FixedCycleCounter
%CyclesTransactionalAborted  = ((CyclesInTxOnly - CyclesInTxCP) / CyclesTotal) * 100.0
%CyclesTransactional = (CyclesInTx / CyclesTotal) * 100.0
%CyclesNonTransactional = 100.0 - %CyclesTransactional
```

CyclesTransactional がほぼ 0 の場合、アプリケーションはロックベースの同期を使用していないか、インテル® TSX 命令を通してロックの無効化に対応した同期ライブラリーを使用していない。後者の場合、インテル® TSX 対応の同期ライブラリーを使用すべきである (「12.3 インテル® TSX 対応の同期ライブラリーの開発」を参照)。

CyclesTransactionalAborted が CyclesTransactional に対して小さい場合、トランザクションの成功率が高く、さらなるチューニングは必要ない。

CyclesTransactionalAborted が CyclesTransactional とほぼ同じで小さくない場合、ほとんどのトランザクション領域はアボートし、ロックの無効化による利点は得られない。この場合の次のステップは、トランザクション・アボートの原因を特定して減らすことである (「12.2.3 トランザクション・アボートを最小限に抑える」を参照)。

12.4.2 無効化するロックを特定してすべてのロックが無効化されることを確認する

このステップは、トランザクション実行で費やされたサイクルが少ない場合に有効である。これは、無効化されるロックが少ないことが原因の可能性がある。MEM_UOPS_RETIRED.LOCK_LOADS イベントをカウントし、RTM_RETIRED.START イベントまたは HLE_RETIRED.START イベントと比較すべきである。ロックのロード数が、開始されたトランザクション領域の数よりもかなり大きい場合、すべてのロックが無効化対象としてマークされていないことが考えられる。MEM_UOPS_RETIRED.LOCK_LOADS の PEBS バージョンでサンプリングを行い、無効化対象になっていないロックを特定できる。ただし、この手法は無効化対象になっていないメタロックを迅速に検出するのには効果的でない (「12.3.7 インテル® TSX を使用するアプリケーション固有のメタロックの無効化」を参照)。また、MEM_UOPS_RETIRED.LOCK_LOADS イベントのコールグラフをプロファイリングすることで、アプリケーション・レベルのクリティカル・セクションをトランザクション実行するためにインテル® TSX を使用すべき高レベルの同期ライブラリーを特定できる。

12.4.3 トランザクション・アボートのサンプリング

ハードウェア実装は、トランザクション・アボート (HLE の場合は HLE_RETIRED.ABORTED、RTM の場合は RTM_RETIRED.ABORTED) をサンプリングするため PEBS プリサイスイベントを定義しており、実行中のすべてのトランザクション・アボートを正確にプロファイルすることができる。PEBS を有効にしてサンプリングし、トランザクション・アボートが発生するコード位置を特定する。PEBS ハンドラー (プロファイリング・ツールの一部) は、PEBS レコードの EventingIP フィールドを用いてトランザクション・アボートの正確なコード位置を報告する。

次のステップでは最も一般的なトランザクション・アボートについて検証し対処する。トランザクション・アボートのサンプリングによって追加のアボートが発生することはない。

12.4.4 プロファイリング・ツールを利用してアボートを分類する

トランザクション・アボートのプロファイリングにより生成される PEBS レコードには、トランザクション・アボートの原因に関する追加情報を示す TX Abort Information フィールドがある。TX Abort Information の下位 32 ビットは Cycles_Last_TX と呼ばれ、アボート前の最後のトランザクション領域で費やされたサイクル数を示す。このデータからトランザクション・アボートのおよそのコストが分かる。

RelativeCostOfAbortForIP = SUM(Cycles_Last_TX_For_IP)

トランザクション・アボートにはパフォーマンスを低下させないものもあれば、パフォーマンスに大きく影響するものもある。プログラマーは、この情報を基にどのトランザクション・アボートに注目すべきかを判断できる。

PEBS レコードの詳細については、『Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B』(英語) の 18.10.5.1 節を参照のこと。

アボートを分類できるように、プロファイリング・ツールはアボートコストを表示できなければならない。

チューニングの推奨事項 22: 最もコストの高いアボートを最初に検証する。

チューニングの推奨事項 23: TX Abort Information にはトランザクション・アボートに関する追加情報が含まれている。

PEBS レコードの **Instruction_Abort** ビット (ビット 34) が設定されている場合、トランザクション・アボートの原因を命令に直接関連付けることができる。これによりアボートに対して、PEBS レコードはトランザクション・アボートの原因となった命令アドレスを記録する。ページフォルト (通常プログラムを終了させるものやプログラム起動時のワーキングセットでフォルトになるものを含む) などの例外もこのカテゴリーに含まれる。

PEBS レコードの **Non_Instruction_Abort** ビット (ビット 35) が設定されている場合、アボートの原因は PEBS レコードで報告された命令アドレスの命令ではない可能性がある。例えば、ほかのスレッドとの間でデータ競合が発生した場合が考えられる。この場合、**Data_Conflict** ビット (ビット 37) も設定される。別の例として、トランザクション実行する読み取りセット/書き込みセットの処理能力の制限によるトランザクション・アボートが挙げられる。これは、Capacity_Write (ビット 38) フィールドと Capacity_Read (ビット 39) フィールドに記録される。

データ競合によるアボートは、トランザクション領域内のどの関数でも発生する可能性がある。そのため、クリティカル・セクション全体にわたって競合の原因を調査したほうが良い。PEBS によって報告される **EventingIP** ではなく、リターン IP (アボートコードの IP) とコールグラフに注目すべきである。リターン IP はロックがインライン展開されていない限り、通常同期ライブラリーを指しているため、呼び出し元からクリティカル・セクションを特定できる。

処理能力が原因の場合、クリティカル・セクション全体にわたってメモリー使用量を減らすように変更する必要があるので、クリティカル・セクション全体 (ReturnIP のプロファイリング) を調査すると良い。

チューニングの推奨事項 24: 命令のアボートは早期に分析すべきだが、プログラム起動後に発生するコストの高いもののみ分析する。

チューニングの推奨事項 25: データ競合や処理能力の制限によるアボートは、アボート時に報告される命令アドレスだけでなく、クリティカル・セクション全体を調査する。

チューニングの推奨事項 26: プロファイラーは、命令が原因でないアボートイベントでは ReturnIP とコールグラフの表示を、命令が原因のアボートイベントでは EventingRIP の表示をサポートしなければならない。

チューニングの推奨事項 27: プロファイリング・ツールはすべての PEBS TX Abort Information ビットを表示できなければならない。

12.4.5 RTM フォールバック・ハンドラーの XABORT 引数

RTM ベースのトランザクション領域のアボートに XABORT 命令が使用されると、EAX レジスターを介してフォールバック・ハンドラーに命令オペランドが渡される。この情報は、RTM 用の PEBS ベースのプロファイリング・ツールでも提供される。プロファイリング・ツールはこの情報を使用してさまざまな XABORT ベースのトランザクション・アボートを分類できる。アボートステータスを定義することは、優れたフォールバック・ハンドラーを記述する上でも役立つ。次の表に、本書で使用するアボートステータスの定義を示す。

表 12-1 RTM アボートステータスの定義

XABORT コード	説明
0xff	テスト時にロックがフリーでなかったことが原因の XABORT ベースのアボート (「12.3.4 ロック無効化に RTM を使用するラッパーの例」を参照)
0xfe	無効化されたロックの値がテストされたことが原因の XABORT ベースのアボート (「12.3.9 RTM ベースのライブラリーで無効化されたロックの値を読み取る」を参照)
0xfd	入れ子の try-lock 内で発生した XABORT ベースのアボート (「12.3.9 RTM ベースのライブラリーで無効化されたロックの値を読み取る」を参照)
0xfc: 0xf0	予約済み

チューニングの推奨事項 28: プロファイリング・ツールは RTM アボートコードを表示できなければならない。

12.4.6 トランザクション・アボートのコールグラフ

プロファイリング・ツールは、パフォーマンス監視情報を収集する際に割り込みをかける。この割り込みはトランザクション・アボートの原因になる。つまり、プロファイリング・ツールはトランザクション・アボートが発生した後にのみ情報を収集することが可能であり、トランザクション領域内で発生したスタック上の関数呼び出しは把握できず、トランザクション実行の開始時のコールグラフのみ見ることができる。PEBS でトランザクション・アボートをサンプリングする場合、RIP フィールドにはアボート後の命令ポインターが、EventingIP フィールドにはアボート時のトランザクション領域内の命令ポインターが含まれる。すべてのサンプリングがトランザクション・アボートの原因となるため、非アボートイベントのサンプリングでも同じことが言える。

アボートの種類に応じて、ReturnIP または EventingIP のいずれかをプロファイリングすると良い。プロファイリング・ツールによって収集されるスタック・コールグラフは常に ReturnIP と関連付けられている。この情報と EventingIP を組み合わせると、トランザクション領域内に関数呼び出しが含まれず、連続していないように見えることがある (EventingIP は最下位の呼び出し元と関連付けられていない可能性がある)。アボートの原因を理解するためトランザクション領域内の関数呼び出しに関する情報が必要な場合は、LBR (最後の分岐レコードの略、12.4.7 節を参照) またはインテル® Software Development Emulator (インテル® SDE) (「12.4.8 インテル® SDE によるインテル® TSX ソフトウェアのプロファイリングとテスト」を参照) を利用できる。

チューニングの推奨事項 29: プロファイラーは ReturnIP と EventingIP を表示できなければならない。

チューニングの推奨事項 30: スタック・コールグラフは常に ReturnIP に関連付けられており、EventingIP と一緒に見た場合、連続していないように見えることがある。

チューニングの推奨事項 31: トランザクション領域内の関数呼び出しを確認するには LBR またはインテル® SDE を利用する。

12.4.7 LBR とトランザクション・アボート

LBR (『Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B』(英語) の 17.4 節を参照) は、トランザクション実行とアボートに関する情報を提供する。一般に LBR の使用法にはインテル® TSX と互換性がある。通常のコールグラフが利用できない場合、LBR を使用することでトランザクション内の情報が得られる。lcall フィルターをコールグラフの代わりに使用できる。ただし、LBR コールグラフ・スタック (『Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B』(英語) の 17.8 節) はインテル® TSX と互換性がなく、完全な情報が得られないことがある。

チューニングの推奨事項 32: PEBS プロファイリング・ハンドラーはアボート時に LBR をサンプリングし、その結果を報告できなければならない。

12.4.8 インテル® SDE によるインテル® TSX ソフトウェアのプロファイリングとテスト

インテル® Software Development Emulator (インテル® SDE) ツール (http://software.intel.com/en-us/articles/intel-software-development-emulator) により、ハードウェアに実装される前に新しい命令セット拡張をソフトウェア開発で利用できる。このツールは、新しい命令を利用するソフトウェアの広範なテスト、デバッグ、解析にも役立つ。

インテル® SDE の各種機能によって、インテル® TSX 命令を使用するプログラムの機能テスト、プロファイル、デバッグを行える。このツールは一般的なトランザクション・アボートの詳細な情報と、ハードウェアで直接利用できない追加のプロファイリング機能をもたらす。エミュレーションによる非常に大きなオーバーヘッドが発生するため、このツールでランタイムおよび絶対パフォーマンス特性を得るべきではない。

「12.4.4 プロファイリング・ツールを利用してアボートを分類する」で述べたとおり、アボートの原因がデータ競合やリソースの制限によるものでない限り、ハードウェアはアボートの原因となった命令の正確なアドレスを報告する。インテル® SDE は正確な命令アドレスに加えて、アプリケーションのソースコード位置、ソースファイル名、行番号、コールスタック、データアドレスの情報も提供する。

次のオプションを指定することで、これらの情報を得られる。

```
-hsw -hle_enabled 1 -rtm-mode full -tsx_stats 1 -tsx_stats_call_stack 1
```

フォールバック・ハンドラーは EAX レジスターからアボートの原因を判断できる。インテル® SDE ツールにエミュレーター・パラメーターとしてEAX レジスター値を渡すと、トランザクション・アボートを強制できるため、開発者はさまざまな EAX 値でフォールバック・ハンドラーをテストできる。すべての RTM ベースのトランザクション実行は、パラメーターとして渡された EAX レジスター値で直ちにアボートする。これは、未解決のページフォルトや同様の操作が原因でトランザクション実行がアボートするケース (EAX = 0) の機能テストにおいて効果的である。

次のオプションを指定することで、これらの情報を得られる。

```
-hsw -rtm-mode abort -rtm_abort_reason EAX
```

12.4.9 HLE 固有のパフォーマンス監視イベント

インテル® TSX のパフォーマンス・イベントには HLE 固有のトランザクション・アボート条件が含まれている。これらのイベントは、「12.2.3.4 HLE 固有のトランザクション・アボート」に示す原因のアボートを追跡する。多くの場合、これらのアボートは同期ライブラリーの実装の問題により発生する。インテル® TSX 対応の同期ライブラリーでは、これらのイベントを測定してその値が無視できるくらいになるまでライブラリーを改善すると良い。

TX_MEM.ABORT_HLE_STORE_TO_ELIDED_LOCK は、XRELEASE プリフィクスを持たないストア操作が、無効化バッファーで無効化されたロックの操作を行ったために発生したトランザクション・アボートの数をカウントする。これは多くの場合、ロックの解放命令に XRELEASE プリフィクスがないことが原因である。

TX_MEM.ABORT_ELISION_BUFFER_NOT_EMPTY は、トランザクション実行をコミットする XRELEASE プリフィクスが付加されたロックの解放命令が、無効化されたロックを持つ無効化バッファーを見つけたために発生したトランザクション・アボートの数をカウントする。これは多くの場合、無効化されなかった (つまり、無効化バッファーにない) ロックに対して XRELEASE を実行するコードシーケンスで発生する。

TX_MEM.ABORT_HLE_ELISION_BUFFER_MISMATCH は、XRELEASE ロックが無効化バッファーのアドレスと値の条件を満たさないために発生したトランザクション・アボートの数をカウントする。これは、例えば XRELEASE 操作によって書き込まれる値が、同じロックに対する XACQUIRE 操作で読み取られた値と異なる場合に発生する。

TX_MEM.ABORT_HLE_ELISION_UNSUPPORTED_ALIGNMENT は、トランザクション領域の読み取りが無効化バッファーのロックにアクセスしたが、読み取れなかったために発生したトランザクション・アボートの数をカウントする。これは通常、アクセスが適切にアライメントされていないか、アクセスが部分的にオーバーラップしているか、あるいは読み取り操作のリニアアドレスが無効化されたロックと異なるが物理アドレスは同じ場合に発生する。これらのイベントは非常にまれである。

12.4.10 インテル® TSX の有用な指標を計算する

ここでは、パフォーマンス・イベントを使って有用な指標を計算する式を示す。イベントのカウントをそのまま利用できることもあるが、場合によってはカウンターのデータを基に計算が必要になる。

次の式は、HLE または RTM トランザクション実行が開始された回数を計算する。ここでは、すべての入れ子の領域を 1 つの領域にまとめている。

```
#HLE Regions Started: HLE_RETIRED.COMMIT + HLE_RETIRED.ABORTED
#RTM Regions Started: RTM_RETIRED.COMMIT + RTM_RETIRED.ABORTED
```

次の式は、アボートした HLE または RTM トランザクション実行の割合を計算する。

```
%AbortedHLE = 100.0 * (HLE_RETIRED.ABORTED / HLE_RETIRED.START)
%AbortedRTM = 100.0 * (RTM_RETIRED.ABORTED / RTM_RETIRED.START)
```

次の式は、トランザクション領域で費やされたサイクル数の平均を計算する (CyclesInTX の計算については「12.4.1 トランザクションの成功を測定する」を参照)。

AvgCyclesInHLE = CyclesInTX / HLE_RETIRED_START
AvgCyclesInRTM = CyclesInTX / RTM_RETIRED.START
AvgCyclesInTX = CyclesInTX / (HLE_RETIRED.START + RTM_RETIRED.START)

次の式は、データ競合によりアボートした HLE または RTM トランザクション実行の割合を計算する。

%AbortedHLEDataConflict = TX_MEM.ABORT_CONFLICT / HLE_RETIRED.START
%AbortedRTMDataConflict = TX_MEM.ABORT_CONFLICT / RTM_RETIRED.START
%AbortedTXDataConflict = TX_MEM.ABORT_CONFLICT / (HLE_RETIRED.START + RTM_RETIRED.START)

次の式は、トランザクション・ストアのリソースの制限によりアボートした HLE または RTM トランザクション実行の数を計算する。

%AbortedTXStoreResource = TX_MEM.ABORT_CAPACITY_WRITE

次の式は、リソースの制限によりアボートした HLE または RTM トランザクション実行の合計数を計算する。L1データキャッシュから追い出されたトランザクション読み取りは、直ちにアボートにならない可能性があるため区別される。

%AbortedHLEResource = HLE_RETIRED.ABORTED_MISC1 - TX_MEM.ABORT_CONFLICT
%AbortedRTMResource = RTM_RETIRED.ABORTED_MISC1 - TX_MEM.ABORT_CONFLICT
%AbortedTXResource = (HLE_RETIRED.ABORTED_MISC1 + RTM_RETIRED.ABORTED_MISC1) - TX_MEM.ABORT_CONFLICT

HLE_RETIRED.ABORTED_MISC1 は、「12.4.9 HLE 固有のパフォーマンス監視イベント」で示したいくつかのイベントの影響を受けることがある。正確な結果を得るためには、まずこれらを最小限に抑えるようにロック・ライブラリーをチューニングする必要がある。

12.5 パフォーマンス・ガイドライン

第 4 世代インテル® Core™ プロセッサーはインテル® TSX をサポートする最初のプロセッサーである。トランザクション実行には実装固有のオーバーヘッドが伴う。パフォーマンスは、将来のマイクロアーキテクチャーで改善される可能性がある。インテル® TSX の初期実装はアプリケーションのクリティカル・セクションにおける一般的な用途を想定している。そのため、このようなオーバーヘッドは相殺され、通常アプリケーション・レベルのパフォーマンスには影響しない。

しかし、考慮すべきいくつかのガイドラインがある。

チューニングの推奨事項 33: インテル® TSX はクリティカル・セクション向けに設計されているため、XBEGIN/XEND 命令と XACQUIRE/XRELEASE プリフィクスのレイテンシーは、LOCK プリフィクス命令のレイテンシーと一致するように意図されている。これらの命令のレイテンシーは通常のロード操作とは異なることに注意する。

トランザクション領域の実行には実装固有のオーバーヘッドがある。そのほとんどは固定コストで、残りはさまざまな動的コンポーネントによるものである。このオーバーヘッドはクリティカル・セクションのサイズやメモリー使用量とはほとんど関係なく、通常マイクロアーキテクチャーのアウトオブオーダー実行によって相殺される。しかし、第 4 世代インテル® Core™ プロセッサー実装では、特定のシーケンスでこのオーバーヘッドが大きくなる可能性がある。特にクリティカル・セクションが非常に小さく、タイトなループ内にある場合 (例えば、マイクロベンチマークで行われる処理など)、オーバーヘッドが大きくなる。実際のアプリケーションでは、通常このような動作は見られない。

このオーバーヘッドは大きなクリティカル・セクションでは相殺されるが、非常に小さなクリティカル・セクションでは相殺されない。オーバーヘッドを減らす簡単なアプローチの 1 つは、クリティカル・セクションの早期にトランザクション・キャッシュラインへアクセスすることである。

12.6 デバッグ・ガイドライン

インテル® TSX を使用するロック無効化の実装はアプリケーションのセマンティクスを変更しない。つまり、アボートされたトランザクション実行中に更新されたすべてのアーキテクチャー・ステートは、ハードウェアによって自動的に破棄される。アプリケーションにトランザクション実行でのみ実行される新しいコードパスを追加する際は注意が必要である (「12.2.4 トランザクション実行専用のコードパスの使用」を参照)。

ただしロックの無効化では、データ競合が起こった場合のみスレッド間の通信が発生するため、スレッド間の実行タイミングが変わり、ロックが通常よりも速く行われるように見えることがある。このタイミングの違いはアプリケーションに潜在的な問題をもたらす可能性がある。この潜在的な問題はインテル® TSX 固有のものではなく、新しい世代のすべてのハードウェアで見られる。

コードのインストルメンテーションは、マルチスレッド・ソフトウェアのデバッグでよく使用される手法である。タイミング関連の問題をデバッグする場合と同様に、コードのインストルメンテーションを行う際は、タイミングを大きく変えたり不要なアボートを引き起こさな

いように注意が必要である。スレッドごとにバッファーを利用して、実行をトレースし特定のイベントを記録できる。タイムスタンプの取得には RDTSC 命令を使用できる。バッファーの出力はクリティカル・セクション外で行うべきである。

トランザクション・アボートは、トランザクション領域内の更新されたメモリー状態をすべて破棄する。この情報はインストルメンテーションでなければトレースできない。トランザクション領域内の問題は、プロファイリング・ツールではトランザクション・アボートとして検出され、LBR 情報から制御フローを再構成できる。

通常の assert() 関数はトランザクション・アボートになり、その出力情報はトランザクション領域外では利用できない。RTM 命令を使用することで assert 関数は、トランザクション実行を終了し、その影響を可視化して、assert 関数でプログラムを終了することができる。次に例を示す。

```
assert(x) => if (!(x)) { while (_xtest()) _xend();  assert(0); }
```

12.7 インテル® TSX 用の一般的な組込み関数

新しいアセンブラー (GNU* binutils 2.23、Microsoft* Visual Studio* 2012) はインテル® TSX 命令をサポートしている。以前のツールではインテル® TSX 命令をバイト値として表現できる。

12.7.1 RTM C 組込み関数

新しい C/C++ コンパイラー (gcc* 4.8、Microsoft* Visual Studio* 2012、インテル® C++ コンパイラー XE 13.0) は、**immintrin.h** ヘッダーファイルで RTM 組込み関数をサポートしている。RTM は新しい命令セットであり、CPUID 命令で RTM 機能フラグを確認してから使用すべきである (『Intel® Architecture Instruction Set Extensions Programming Reference』(英語) の第 8 章を参照)。

_xbegin()

_xbegin() はトランザクション領域を開始して、トランザクション領域に入ると _XBEGIN_STARTED を返し、そうでない場合はアボートコードを返す。_xbegin() の戻り値が _XBEGIN_STARTED (0 でない値) であることを確認することが重要である。0 はアボートコードである。値が _XBEGIN_STARTED でない場合、リターンコードには、_xabort() によって渡される各種ステータスビットとオプションの 8 ビット定数が含まれる。

以下に、有効なステータスビットを示す。

- _XABORT_EXPLICIT: _xabort() によって発生したアボート。_XABORT_CODE(status) には、_xabort() へ渡された値が含まれる。
- _XABORT_RETRY: このビットが設定されている場合は、再試行によりトランザクション領域をコミットできる可能性がある。設定されていなければ、再試行しても成功する確率が低い。
- _XABORT_CAPACITY: 処理能力の制限によるアボート。
- _XABORT_DEBUG: デバッグトラップによるアボート。
- _XABORT_NESTED: 入れ子のトランザクションで発生したアボート。

_xend()

_xend() はトランザクションをコミットする。

_xtest()

_xtest() は、コードが現在トランザクション実行中の場合は真を返す。HLE でも利用できる。

_xabort()

_xabort(constant) は現在のトランザクションをアボートする。constant は8 ビットの定数でなければならない。この定数は _xbegin() によって返されるステータスコードに含まれており、_XABORT_EXPLICIT フラグが設定されている場合、_XABORT_CODE() でアクセスできる。推奨される利用法については 4.5 節を参照のこと。

gcc* 4.8 以降では、-mrtm コンパイラー・オプションを指定してこれらの組込み関数を有効にする必要がある。

12.7.1.1 古い gcc* 互換コンパイラーによる RTM 組込み関数のエミュレート

immintrin.h で RTM 組込み関数をサポートしていない古い gcc* 互換コンパイラーでは、次の等価なインライン・アセンブラーを利用できる。

例 12-11 古い gcc* コンパイラー用にエミュレートされた RTM 組込み関数

```
/* immintrin.h でこのインターフェイスをサポートしている新しいツールでは不要 */
#define _XBEGIN_STARTED         (~0u)
#define _XABORT_EXPLICIT        (1 << 0)
#define _XABORT_RETRY           (1 << 1)
#define _XABORT_CONFLICT        (1 << 2)
#define _XABORT_CAPACITY        (1 << 3)
#define _XABORT_DEBUG           (1 << 4)
#define _XABORT_NESTED          (1 << 5)
#define _XABORT_CODE(x)         (((x) >> 24) & 0xff)

#define __force_inline __attribute__((__always_inline__)) inline
```

```
static __force_inline int _xbegin(void)
{
        int ret = _XBEGIN_STARTED;
        asm volatile(".byte 0xc7,0xf8 ; .long 0" : "+a" (ret) :: "memory");
        return ret;
}
```

```
static __force_inline void _xend(void)
{
         asm volatile(".byte 0x0f,0x01,0xd5" ::: "memory");
}
```

```
static __force_inline void _xabort(const unsigned int status)
{
        asm volatile(".byte 0xc6,0xf8,%P0" :: "i" (status) : "memory");
}
```

```
static __force_inline int _xtest(void)
{
        unsigned char out;
        asm volatile(".byte 0x0f,0x01,0xd6 ; setnz %0" : "=r" (out) :: "memory");
        return out;
}
```

12.7.2 gcc* およびその他の Linux* 互換コンパイラーの HLE 組込み関数

Linux* および互換システムでは、HLE は gcc* 4.8 および古い形式の C11 アトミック・プリミティブの拡張として実装されている。HLE XACQUIRE を使用するにはメモリーモデル引数に __ATOMIC_HLE_ACQUIRE フラグを設定し、HLE XRELEASE を使用するには __ATOMIC_HLE_RELEASE フラグを設定する。

メモリーモデルは、__ATOMIC_HLE_ACQUIRE では __ATOMIC_ACQUIRE 以上、__ATOMIC_HLE_RELEASE では __ATOMIC_RELEASE 以上でなければならない。失敗メモリーモデルと成功メモリーモデルを含む操作 (__atomic_compare_exchange_n など) では、HLE フラグは成功メモリーモデルでのみサポートされる。

HLE は、IA アトミック命令に直接変換可能なアトミック操作でのみサポートされる。次の場合はサポートされない。

- 32 ビットのターゲット上の 8 バイト値
- 16 バイト値
- 加算/減算を除く、結果にアクセスするフェッチ命令または命令フェッチ
- __atomic_store と __atomic_clear は、__ATOMIC_HLE_RELEASE のみサポート

12.7.2.1 gcc* 4.8 による HLE 組込み関数の生成

gcc* 4.8 のいくつかのバージョンでは、コンパイラーの不具合により、アトミック組込み関数を用いて HLE ヒントを生成するには最適化レベル -O2 以上を指定しなければならない。

12.7.2.2 C++11 の <atomic> のサポート

gcc* 4.8 は C++11 の <atomic> ヘッダーをサポートしている。このヘッダーで定義されているメモリーモデルは、C アトミック・インターフェイスに似た HLE フラグで拡張されている。2 つの新しいフラグ __memory_order_hle_acquire と __memory_order_hle_release が定義されており、C アトミック組込み関数の制限が適用される。

以下に C++ の例を示す。

例 12-12 HLE 組込み関数の C++ の例

```
#include <atomic>
#include <immintrin.h>
using namespace std;
atomic_flag lock;
for (;;) {
    if (!lock.test_and_test(memory_order_acquire|__memory_order_hle_acquire) {
        // HLE によるロックの無効化を使用するクリティカル・セクション
        lock.clear(memory_order_release|__memory_order_hle_release);
    break;
    } else {
       // ロックを取得できなかったため待機して再試行する
       while (lock.load())
            _mm_pause();    // ロックがビジーなためトランザクション領域をアボートする
    }
}
```

12.7.2.3 古い gcc* 互換コンパイラーによる HLE 組込み関数のエミュレート

これらの組込み関数をサポートしていない古いコンパイラーではインライン・アセンブリーを利用できる。以下に、__atomic_exchange_n(&lock, 1, __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE) をエミュレートする例を示す。

例 12-13 古い gcc* コンパイラー用にエミュレートされた HLE 組込み関数

```
#define XACQUIRE ".byte 0xf2; " /* XACQUIRE をサポートしていない古いアセンブラー向け */
#define XRELEASE ".byte 0xf3; "
static inline int hle_acquire_xchg(int *lock, int val)
{
    asm volatile(XACQUIRE "xchg %0,%1" : "+r" (val), "+m" (*lock) :: "memory");
    return val;
}
```

```
static void hle_release_store(int *lock, int val)
{
    asm volatile(XRELEASE "mov %0,%1" : "r" (val), "+m" (*lock) :: "memory");
}
```

12.7.3 Windows* C/C++ コンパイラーの HLE 組込み関数

Windows* C/C++ コンパイラー (Microsoft* Visual Studio* 2012 およびインテル® C++ コンパイラー XE 13.0) は、HLE プリフィクスを持つ特定のアトミック組込み関数を提供している。

例 12-14 インテル® コンパイラーと Microsoft* コンパイラーでサポートされている HLE 組込み関数

アトミック比較交換操作:

```
long _InterlockedCompareExchange_HLEAcquire(long volatile *Destination, long Exchange, long Comparand);
__int64 _InterlockedCompareExchange64_HLEAcquire(__int64 volatile *Destination, __int64 Exchange, __int64 Comparand);
void * _InterlockedCompareExchangePointer_HLEAcquire(void * volatile *Destination, void * Exchange, void * Comparand);
long _InterlockedCompareExchange_HLERelease(long volatile *Destination, long Exchange, long Comparand);
__int64 _InterlockedCompareExchange64_HLERelease(__int64 volatile *Destination, __int64 Exchange, __int64 Comparand);
void * _InterlockedCompareExchangePointer_HLERelease(void * volatile *Destination, void * Exchange, void * Comparand);
```

アトミック加算:

```
long _InterlockedExchangeAdd_HLEAcquire(long volatile *Addend, long Value);
__int64 _InterlockedExchangeAdd64_HLEAcquire(__int64 volatile *Addend, __int64 Value);
long _InterlockedExchangeAdd_HLERelease(long volatile *Addend, long Value);
__int64 _InterlockedExchangeAdd64_HLERelease(__int64 volatile *Addend, __int64 Value);
```

HLE プリフィクス・ストア組込み関数:

```
void _Store_HLERelease(long volatile *Destination, long Value);
void _Store64_HLERelease(__int64 volatile *Destination, __int64 Value);
void _StorePointer_HLERelease(void * volatile *Destination, void * Value);
```

組込み関数の詳細については、コンパイラーのドキュメントを参照のこと。