修士論文

排他的および投機的実行による トランザクショナルメモリの高速化

Exclusive and Speculative Control for Compound Operations on Transactional Memory

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学大学院 工学研究科
修士課程 創成シミュレーション工学専攻
平成 25 年度入学　25413558 番

橋本 高志良

平成 27 年 2 月 4 日

排他的および投機的実行によるトランザクショナルメモリの高速化

橋本 高志良

内容梗概

マルチコア環境における並列プログラミングでは，共有リソースに対するメモリアクセスを調停する必要があり，その調停を行う機構として一般的にロックが用いられてきた．しかし，ロック操作にともなうオーバヘッドやデッドロックの発生に起因する並列性の低下，また適切なロック粒度を設定することが困難であるなどの多くの問題があるため，ロックはプログラマにとって必ずしも利用しやすい機構ではない．そこで，ロックを使用しない並行性制御機構としてトランザクショナルメモリ（Transactional Memory: TM）が提案されている．TM は，トランザクションとして定義した一連の命令列を投機的に実行することで，ロックを用いる場合よりも高い並列性を実現できる．

この TM をハードウェア上に実現したものをハードウェアトランザクショナルメモリ（Hardware Transactional Memory: HTM）と呼ぶ．HTM では，各キャッシュラインに対して Read ビットおよび Write ビットという，トランザクション内で発生した Read アクセスおよび Write アクセスの有無を記憶するフィールドが追加されている．そして，各スレッドがキャッシュの一貫性を保つための要求を受け取った際にこれらのビットを参照することで，メモリアクセス競合を検査する．競合が発生した場合，これを検出したスレッドは，競合相手のスレッドがトランザクションの実行を終了するまで，自身の実行するトランザクションを一時的に停止させる．これをストールという．ここで，複数のスレッドが実行トランザクションをストールさせた場合，デッドロック状態に陥ってしまう可能性がある．このデッドロック状態を回避するために，いずれかのトランザクションの途中結果が破棄される．これをアボートという．一方で，トランザクションの終了まで競合が発生しなかった場合は，トランザクション内で更新された値がメモリ上に反映される．これをコミットという．

さて，一般に共有変数に対する Read アクセスは，その後に Write アクセスをともなう場合が多く見られる．具体的には，複合演算子や複合代入式を用いる複合操作などがこれにあたる．HTM において，そのようなアクセスを含むトランザクションを複数のスレッドが並列に実行し，それらのスレッドが当該変数に対する Read アクセスのみを完了した状態となった場合，その後の Write アクセスにより結果として競合が発生してしまう．この競合により，お互いのトランザクションをストールさせることで，いずれかのスレッドがトランザクションをアボートすることになる．この時，ト

ランザクションをアボートしたスレッドの処理は一切進行していないことになるため，当該スレッドと競合していたスレッドのストールは完全に無駄となってしまう．このようなストールは Futile Stall と呼ばれ，これに起因するアボートとともに HTM の全体性能を著しく低下させる要因となりうる．そこで本論文では，同一の共有変数に対する Read，Write の順序でのアクセスを含む処理を排他的に実行する手法を提案する．この手法では，同一変数に対する Read，Write の順序でのアクセスを含むトランザクションを実行するスレッドが，当該変数に対して Read アクセスした場合，そのスレッドは実行トランザクションをコミットするまで，他のスレッドによる当該変数へのアクセスを許可しないようにする．これにより，上述した Futile Stall およびそれに起因するアボートを抑制し，HTM の全体性能を向上させることを目指す．

ここで，一般的なプログラムで実行されるトランザクションには，同一の共有変数に対する Read，Write の順序でのアクセスの後に，当該変数に対する再度のアクセスを含まないものがある．このようなトランザクションを複数のスレッドが並列実行する場合，あるスレッドが同一の共有変数に対する Read，Write の順序でのアクセスを完了した時点で，他スレッドが当該変数に投機的にアクセスしたとしても，結果として一貫性が保たれる可能性がある．そのため，同一の共有変数に対する Read，Write の順序でのアクセスを含む処理を排他実行する手法には，さらなる高速化の余地があると考えられる．そこで本論文では，トランザクション内で同一の共有変数に対する Read，Write の順序でのアクセスを完了した時点で，当該トランザクションのコミットに先立って他スレッドによる投機的アクセスを許可する手法を併せて提案する．

2 つの提案手法の有効性を検証するため，HTM の研究で広く用いられる LogTM を拡張し，シミュレーションによる評価を行った．評価の対象として GEMS microbench，SPLASH-2 および STAMP を用いた結果，1 つめの提案手法では既存の LogTM と比較して 16 スレッド実行時で最大 72.3%，平均 13.6%の高速化を達成し，2 つめの提案手法では既存の LogTM と比較して 16 スレッド実行時で最大 67.2%，平均 13.9%の高速化を達成することを確認した．

排他的および投機的実行によるトランザクショナルメモリの高速化

目次

1　はじめに

これまでのプロセッサ高速化技術は，スーパスカラやSIMDのような命令間の並列性に基づいた多くの高速化手法や，集積回路の微細化による高クロック化の実現によって支えられてきた．しかしながら，プログラム中で抽出できる命令レベル並列性には限界があることや，消費電力や配線遅延の相対的な増大という問題から，単一コアにおける動作クロック周波数の向上も困難となりつつある．この流れを受け，単一チップ上に複数のプロセッサコアを搭載したマルチコアプロセッサが広く普及してきている．マルチコアプロセッサでは，これまで単一コアで実行していたタスクを複数のプロセッサコアがそれぞれ分担して実行することで，単一コアで実行した場合よりもスループットを向上させることができる．

このようなマルチコア環境における並列プログラミングでは，複数のプロセッサコア間で単一アドレス空間を共有する，共有メモリ型並列プログラミングが一般的である．このプログラミングモデルでは，共有リソースであるメモリロケーションに対するアクセスを調停する必要があり，そのための機構として一般的にロックが用いられてきた．しかし，ロックを用いた場合，ロック操作にともなうオーバヘッドやデッドロックの発生に起因する並列性の低下などの問題によって，性能が低下してしまう可能性がある．さらに，プログラムが大規模で複雑であるほど，適切なロック粒度を設定することが困難となる．例えば，粗粒度なロックを用いる場合，プログラムの設計が容易となる一方でクリティカルセクションが大きくなるために並列性が損なわれてしまう．また，細粒度なロックを用いる場合，プログラムの並列性が向上する一方でプログラムの設計自体が複雑なものとなってしまう．したがって，ロックはプログラマにとって必ずしも利用しやすい機構ではない．そこで，ロックを使用しない並行性制御機構としてトランザクショナルメモリ（Transactional Memory: TM）[1] が提案されている．

TMは，従来ロックで保護されていたクリティカルセクションをトランザクションとして定義し，共有リソースであるメモリロケーションに対するアクセスにおいて競合が発生しない限り，トランザクションを投機実行するため，ロックを用いる場合よりも並列性が向上する．なお，TMではトランザクションが投機的に実行されるため，共有リソースであるメモリロケーションに対する更新の際には，更新前あるいは更新後の値を保持しておく必要がある（バージョン管理）．また，トランザクションを実行するスレッド間において，同一のメモリロケーションに対するアクセス競合が発生するか否

かを常に検査する必要がある（競合検出）．このTMをハードウェア上に実現したハードウェアトランザクショナルメモリ（Hardware Transactional Memory: HTM）では，バージョン管理および競合検出のための機構をハードウェアで実現することで，トランザクション処理のためのオーバヘッドを軽減している．このような利点からHTMは現在大きな注目を集めており，IBM社のスーパーコンピュータBlueGene/Q[2]やIntel社のマイクロアーキテクチャHaswell[3]などはTMのためのハードウェアサポートを備えている．

さて，一般に共有変数に対するReadアクセスは，その後にWriteアクセスをともなう場合が多く見られる．具体的には，複合演算子や複合代入式を用いる複合操作などがこれにあたる．HTMにおいて，そのようなアクセスを含むトランザクションを複数のスレッドが並列に実行し，それらのスレッドが当該変数に対するReadアクセスのみを完了した状態となった場合，その後のWriteアクセスにより結果として競合が発生してしまう．この競合により，お互いのトランザクションをストールさせることで，いずれかのスレッドがトランザクションをアボートすることになる．この時，トランザクションをアボートしたスレッドの処理は一切進行していないことになるため，当該スレッドと競合していたスレッドのストールは完全に無駄となってしまう．このようなストールはFutile Stallと呼ばれ，これに起因するアボートとともにHTMの全体性能を著しく低下させる要因となりうる．そこで本論文では，同一の共有変数に対するRead，Writeの順序でのアクセスを含む処理を排他的に実行する手法を提案する．この手法では，同一変数に対するRead，Writeの順序でのアクセスを含むトランザクションを実行するスレッドが，当該変数に対してReadアクセスした場合，そのスレッドは実行トランザクションをコミットするまで，他のスレッドによる当該変数へのアクセスを許可しないようにする．これにより，上述したFutile Stallおよびそれに起因するアボートを抑制し，HTMの全体性能を向上させることを目指す．

ここで，一般的なプログラムで実行されるトランザクションには，同一の共有変数に対するRead，Writeの順序でのアクセスの後に，当該変数に対する再度のアクセスを含まないものがある．このようなトランザクションを複数のスレッドが並列実行する場合，あるスレッドが同一変数に対するRead，Writeの順序でのアクセスを完了した時点で，他スレッドが当該変数に投機的にアクセスしたとしても，結果として一貫性が保たれる可能性がある．そのため，同一変数に対するRead，Writeの順序でのアクセスを含む処理を排他実行する手法には，さらなる高速化の余地があると考えられる．そこで本論文では，トランザクション内で同一変数に対するRead，Writeの順序

でのアクセスを完了した時点で，当該トランザクションのコミットに先立って他スレッドによる投機的アクセスを許可する手法を併せて提案する．この手法により，同一変数に対する Read，Write の順序でのアクセスを含む処理を排他実行する手法のさらなる高速化を目指す．

以下，2 章で研究対象である HTM の概要とその問題点について述べ，3 章で本研究に関連する既存研究について述べる．4 章で 1 つめの提案手法とその実現方法について述べ，5 章で 2 つめの提案手法とその実現方法について述べる．6 章で 2 つの手法をそれぞれ評価し，7 章で結論を述べる．

2　ハードウェアトランザクショナルメモリ

本章では，本研究の対象となる HTM の概要とその問題点について述べる．

2.1　基本概念

マルチコアプロセッサにおける並列プログラミングでは，複数のプロセッサコアが単一のアドレス空間を共有している．このような共有メモリ型並列プログラミングでは，共有リソースであるメモリロケーションに対するアクセスを調停する必要があり，そのための機構として一般的にロックが用いられてきた．しかし，ロックを用いた場合，ロック操作にともなうオーバヘッドやデッドロックの発生に起因する並列性の低下などの問題によって，性能が低下してしまう可能性がある．さらに，プログラムが大規模で複雑であるほど，適切なロック粒度を設定することが困難となる．例えば，粗粒度なロックを用いる場合，プログラムの設計が容易となる一方でクリティカルセクションが大きくなるために並列性が損なわれてしまう．また，細粒度なロックを用いる場合，プログラムの並列性が向上する一方でプログラムの設計自体が複雑なものとなってしまう．以上のような特徴は，ロックを用いたプログラム設計を困難とする要因となっている．そこで，ロックを使用しない並行性制御機構である**トランザクショナルメモリ（Transactional Memory: TM）**が提案されている．TM はデータベースにおけるトランザクション処理で用いられる概念をメモリアクセスに対して適用した機構である．この TM では，従来ロックで保護されていたクリティカルセクションを**トランザクション**として定義し，これを投機的に実行することでロックを用いる場合よりも並列度を向上させている．このトランザクションは以下の 2 つの性質を満たす．

Serializability（直列可能性）:

並行実行されたトランザクションの実行結果は，当該トランザクションを逐次的

に実行した場合と同じであり，全てのスレッドにおいて同一の順序で観測される．

Atomicity（不可分性）:

トランザクションはその操作が完全に実行されるか，もしくは全く実行されないかのいずれかでなければならず，各トランザクション内における操作はトランザクションの終了と同時に観測される．そのため，操作の途中経過が他のスレッドから観測されることはない．

以上の性質を保証するために，TM はトランザクション内で発生するメモリアクセスを監視する．複数のトランザクション内で同一の共有変数に対するアクセスが確認された際に，これらのアクセスがトランザクションの性質を満たさない場合，**競合**が検出される．この操作を**競合検出（Conflict Detection）**という．競合が検出された場合，TM では一方のスレッドがトランザクションの実行を中断させることで競合を回避する．これを**ストール（stall）**という．さらに，複数のスレッドが実行トランザクションをストールさせている状態で，デッドロックの可能性があると判断された場合，いずれかのスレッドがトランザクションの途中結果を破棄する．これを**アボート（abort）**という．その後，トランザクションをアボートしたスレッドはトランザクション開始時点から処理を再実行する．これに対して，トランザクションが終了するまで競合が検出されなかった場合，各スレッドはトランザクション内で更新された値をメモリ上に反映させる．これを**コミット（commit）**という．なお，アボート操作およびコミット操作を行うためには，更新後の新しい値と更新前の古い値を保持しておく必要がある．そのため，トランザクション内で更新された新しい値あるいは更新される前の古い値が，そのアドレスとともに別領域に保持される．このようなデータの管理を**バージョン管理（Version Management）**という．

このように，TM はロックによる排他制御と同等のセマンティクスを維持しつつ，競合が発生しない限りトランザクションを投機実行できる．これにより，ロックを使用する場合よりもプログラムの並列性が向上するため，コア数に応じた性能のスケールが期待できる．また，プログラマはデッドロックの発生を考慮する必要がなくなるため，クリティカルセクションを含む処理領域をトランザクションとして定義するだけで，容易に並列プログラムを構築できる．なお，TM で行われる競合検出とバージョン管理の機能はハードウェア上またはソフトウェア上に実装される．このうち，ハードウェア上に実装された TM を**ハードウェアトランザクショナルメモリ（Hardware Transactional Memory: HTM）**と呼ぶ．一般的に，HTM は更新後の新しい値と更新前の古い値を併存させるために，一方を書き込み先のキャッシュ上に保持し，もう

一方をこれとは別の領域に退避させる．また，競合を検出および解決する機構をハードウェアによりサポートしているため，速度性能が高い．このような利点からHTMは現在大きな注目を集めており，IBM社のスーパーコンピュータBlueGene/Q[2]やIntel社のマイクロアーキテクチャHaswell[3]などはTMのためのハードウェアサポートを備えている．これに対して，ソフトウェア上に実装されたTMをソフトウェアトランザクショナルメモリ（STM）[4]と呼ぶ．STMは，HTMのようなハードウェア拡張は必要ないが，TM上で行われる操作が全てソフトウェアにより実現されるため，オーバヘッドが大きい．そこで本論文では，速度性能の高いHTMを研究の対象とする．

2.2 データのバージョン管理

トランザクションの投機実行では，実行結果が破棄される可能性があるため，更新前の古い値と更新後の新しい値を併存させる必要がある．そこでHTMでは，トランザクション内で更新された新しい値，もしくは更新前の古い値がそのアドレスとともに別の領域に保持される．このようなデータのバージョン管理は，以下の2つの方式に大別される．

Eager Version Management:

書き換え前の古い値を別領域に退避させ，新しい値をメモリに上書きする．コミットはバックアップ領域に退避された値を破棄するだけであるため，高速に行うことができるが，アボート時には退避された値をメモリにリストアする必要がある．

Lazy Version Management:

書き換え前の古い値をメモリに残し，新しい値を別領域に記憶する．アボートは高速に行うことができるが，コミット時にメモリに値をコピーする必要がある．

Eager方式は，必ず実行されるコミットを高速に処理し，必ずしも発生するとは限らないアボートにコストを払う考え方である．そのため，アボートが繰り返し発生してしまうようなプログラムでは不利となる場合もある．しかし，Lazy方式ではコミットのためのオーバヘッドは削減の余地がほぼないのに対し，Eager方式ではスケジューリングの改良により，競合やアボートの発生自体を抑制することで性能向上できる余地が大きいと考えられる．したがって本論文では，Eager方式を研究対象とする．

ここで，このEager方式におけるバージョン管理の動作を**図1**を用いて説明する．なお，図中のMemoryおよびBackupはそれぞれメモリ，バックアップ領域を示す．まず，共有変数のアドレスであるAに値10が格納されている状態でトランザクションの実行が開始されたとする．その後，この状態からトランザクションの実行が進み，A

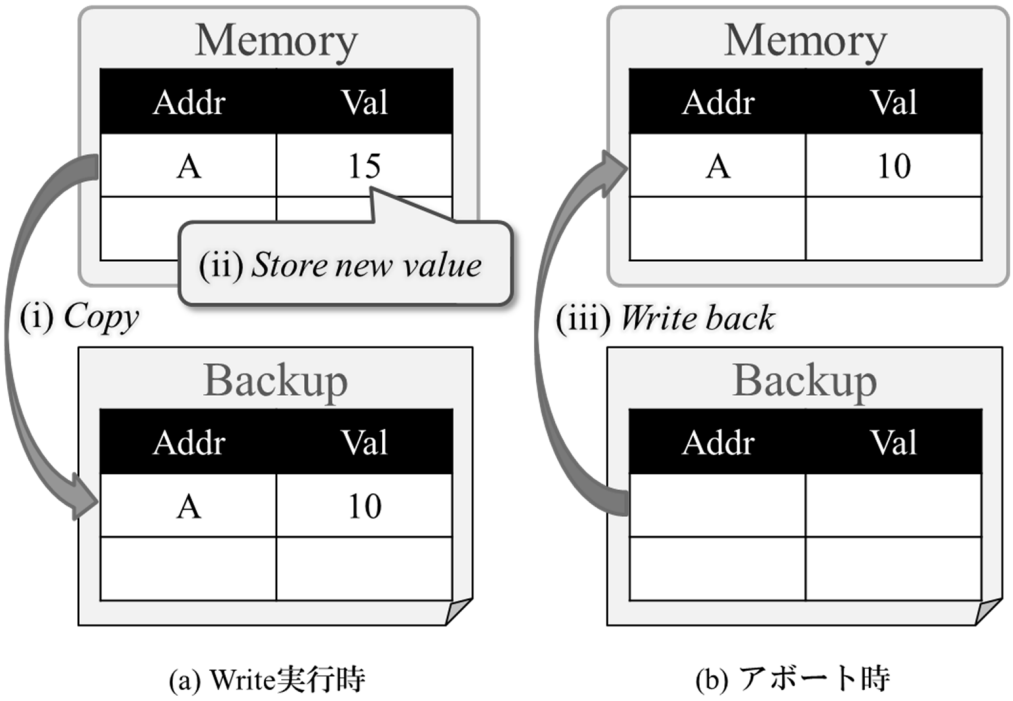


図 1: Eager 方式におけるバージョン管理

に値 15 が書き込まれるとすると，図 1(a) に示すように Write アクセス対象のアドレスと，書き換え前の値である 10 がメモリからバックアップ領域に退避される (i)．一方，メモリ上には Write アクセスの実行結果である値 15 が上書きされる (ii)．

次に図 1(a) の状態からさらに実行が進み，投機実行が成功した場合は，トランザクションがコミットされる．この時，Write の実行結果である値 15 は既にメモリ上に反映されているため，バックアップ領域に退避したアドレスおよび値を破棄することで，トランザクション内で行われた変更を確定させる．一方で投機実行が失敗した場合，トランザクションはアボートされる．アボート時には，図 1(b) に示すように，バックアップ領域に退避されたデータが元のメモリアドレスに書き戻される (iii)．これにより，トランザクション開始時点のメモリ状態を復元することができる．また，アボート後にトランザクションを再実行するためには，メモリと同様にレジスタもトランザクション開始時の状態に戻す必要がある．これを実現するため，HTM ではトランザクション開始時にその時点におけるレジスタの状態を取得し，その状態をバックアップ領域に保存しておく．そして，アボート時にバックアップ領域を参照しトランザクション開始時点のレジスタの状態を復元する．

2.3 競合の検出と解決

トランザクションの投機実行において，競合を検出するためにはどのアドレスがトランザクション内でアクセスされたかを記憶する必要がある．そのため，HTM では一般的に，各キャッシュラインに対して **Read ビット（R ビット）**および **Write ビット（W ビット）**と呼ばれるフィールドが追加されている．各ビットはトランザクション内で当該ラインに対する Read アクセスおよび Write アクセスが発生した場合にセットされ，トランザクションのコミットおよびアボート時にクリアされる．

これらのビットを操作するために，HTM ではキャッシュの一貫性を保持するプロトコルであるディレクトリベース [5] の Illinois プロトコル [6] を拡張している．この一貫性プロトコルでは，スレッドがあるメモリアドレスにアクセスする際，キャッシュの状態を管理するディレクトリ機構に対し，既に同一のアドレスがアクセスされているかを問い合わせるリクエストが送信される．ディレクトリはこの際，過去に当該アドレスへアクセスしたスレッドに対してリクエストを転送する．拡張したプロトコルにおいて，各スレッドはこのリクエストを受信すると，キャッシュに追加された R および W ビットを参照する．これにより，他のスレッドとの競合を監視する．この時，以下の 3 パターンのアクセスが発生した場合に競合が検出される．

Read after Write(RaW):

あるトランザクション内で Write アクセス済みであるアドレスに対して，他のトランザクションを実行するスレッドから Read アクセスされるパターン．つまり，W ビットがセットされているアドレスに対する Read リクエストが検出された場合である．この Read リクエストが許可されると，トランザクションがコミットされる前に，トランザクション内で更新した値が他のトランザクションを実行するスレッドにより読み出されることで，異なった結果となる可能性があるため，トランザクションの性質を満たさなくなる．

Write after Read(WaR):

あるトランザクション内で Read アクセス済みであるアドレスに対して，他のトランザクションを実行するスレッドから Write アクセスされるパターン．つまり，R ビットがセットされているアドレスに対する Write リクエストが検出された場合である．この Write リクエストが許可されると，トランザクションがコミットされる前に，トランザクション内で読み出した値が他のトランザクションを実行するスレッドにより更新されることで，異なった結果となる可能性があるため，トランザクションの性質を満たさなくなる．

Write after Write(WaW):

あるトランザクション内でWriteアクセス済みであるアドレスに対して，他のトランザクションを実行するスレッドからWriteアクセスされるパターン．つまり，Wビットがセットされているアドレスに対するWriteリクエストが検出された場合である．このWriteリクエストが許可されると，トランザクションがコミットされる前に，トランザクション内で更新した値が他のトランザクションを実行するスレッドによって更新されることで，異なった結果となる可能性があるため，トランザクションの性質を満たさなくなる．

以上のようなアクセスパターンが検出されると，競合が発生したと判断し，競合を検出したスレッドからリクエストを送信したスレッドに対して *Nack* が返信される．これにより，*Nack* を受信したスレッドは自身の試みたアクセスにより競合が発生したことを知ることができる．なお，この競合検出方式は，競合検査のタイミングによって以下の2つに大別される．

Eager Conflict Detection:

トランザクション内でメモリアクセスが発生した時点で，そのアクセスに関する競合が発生したか否かを検査する．

Lazy Conflict Detection:

トランザクションのコミットを試みる時点で，そのトランザクション内で行われたすべてのアクセスに関して競合が発生したか否かを検査する．

Eager方式ではメモリアクセスごとに競合の有無を検査するため，競合が多く発生するプログラムではストールが頻繁に発生してしまう可能性がある．一方，Lazy方式ではトランザクションのコミット時に競合の有無を検査するため，競合が多く発生するプログラムであってもストールが頻繁には発生しない．しかし，Lazy方式では実際に競合が発生してからそれを検出するまでの時間が長くなるため，無駄な実行が増大して効率が悪くなってしまう．これに対し，Eager方式ではスケジューリングの改良により，競合やアボートの発生自体を抑制することで性能向上できる余地が大きいと考えられる．したがって本論文では，Eager方式を本研究の対象とする．

ここで，Eager方式において競合が発生した場合，次の2つの方法のいずれかにより競合が解決される．1つめの方法は，競合時に即座に片方のトランザクションをアボートすることで競合を解決するというものである．しかしこの方法では，アボートが頻繁に引き起こされる可能性がある．2つめの方法は，競合時に一方のトランザクションがコミットもしくはアボートされるまで，もう一方のトランザクションを一時

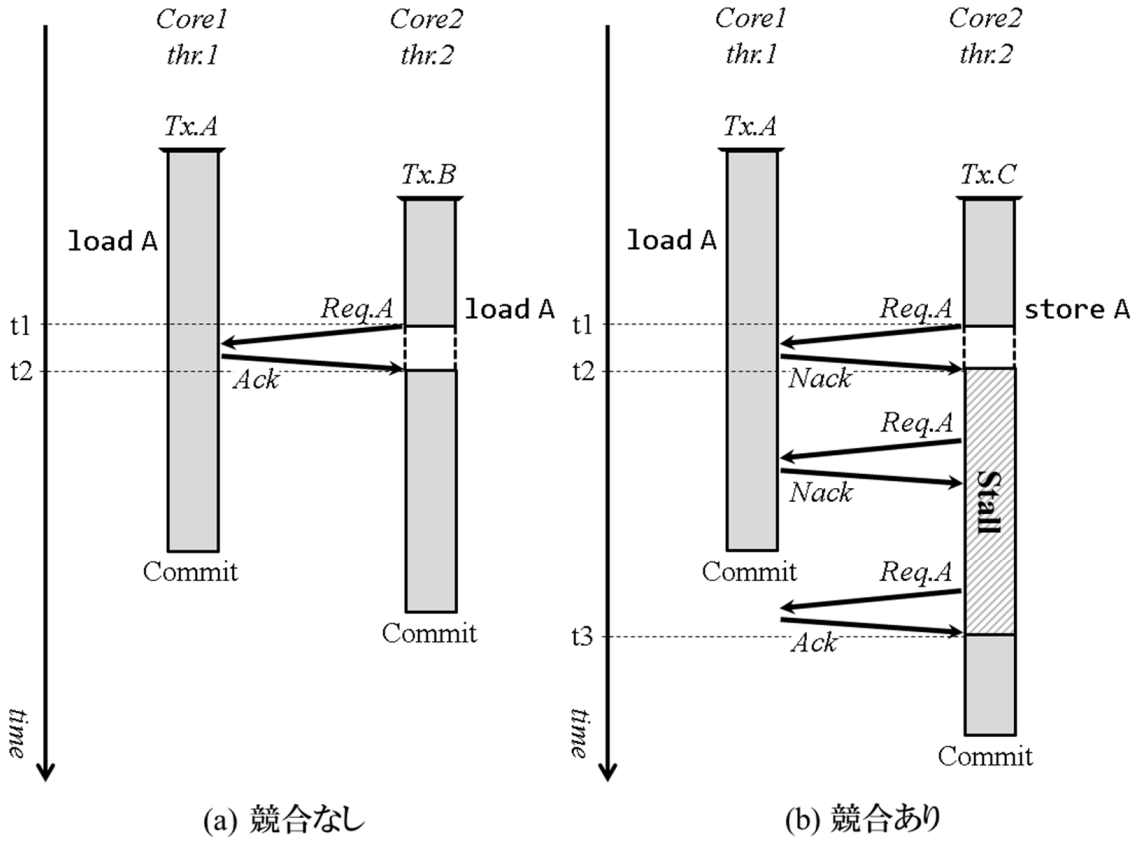


図 2: 競合の検出

的にストールさせることで競合を解決するというものである．この方法は，より多くのトランザクションを並列実行させることができるため，一般的な HTM で多く採用されている．なお，Eager 方式において競合が検出されトランザクションをストールさせたスレッドは，競合アドレスに対するリクエストを定期的に送信し続けることで，競合相手スレッドの実行トランザクションが終了したか否かを監視する．一方で，競合が検出されなかった場合は，従来の一貫性プロトコルにしたがって処理が進む．例えば，無効化リクエストに対しては *Ack* が返信され，読み出しのための共有リクエストに対しては *Ack* とともに共有されるデータが返信される．

ここで，並列に実行される 2 つのトランザクションの間で発生する競合を Eager 方式で検出する際の動作モデルを**図 2** に示す．なお，図 2 では *Core1* および *Core2* はそれぞれコアを，*thr.1* および *thr.2* は各コア上で動作するスレッドを示し，それらのスレッドはそれぞれ異なるトランザクションを実行しているとする．また，図 2 中の A は共有変数のアドレスであるとし，共有メモリおよびディレクトリと各コア間における通信は図中では省略している．まず，競合が発生しない場合について，2 つのスレッ

ド *thr.1* および *thr.2* がそれぞれトランザクション *Tx.A* および *Tx.B* を実行する図 2(a) の例を用いて，その動作を説明する．この図 2(a) において，*thr.1* が `load A` を実行した後に，*thr.2* が `load A` の実行を試みたとする（時刻 t1）．この時，*thr.2* は A に対するアクセスリクエスト *Req.A* を *thr.1* に送信する．この *Req.A* を受信した *thr.1* は，*thr.2* が A にアクセスしようとしていることを知るが，このような Read after Read アクセスは，許可したとしてもトランザクションが持つべき性質を損なわないため，競合としては検出されない．したがって，*thr.1* は *thr.2* に対して *Ack* を返信する．一方，これを受信した *thr.2* は `load A` を実行できる（t2）．

これに対し，競合が発生する場合について，2 つのスレッド *thr.1* および *thr.2* がそれぞれトランザクション *Tx.A* および *Tx.C* を実行する図 2(b) の例を用いて，その動作を説明する．この例では図 2(a) の例と同様に，まず *thr.1* が `load A` を実行する．その後，*thr.2* が同一アドレスに対して Write アクセスを試みたとする（時刻 t1）．この時，*thr.2* は *thr.1* に *Req.A* を送信するが，これは競合を引き起こすアクセスパターンの 1 つである WaR アクセスに該当する．したがって，*thr.2* から *Req.A* を受信した *thr.1* は，競合が検出されたことを通知するために *Nack* を返信する．一方，この *Nack* を受信した *thr.2* は *Tx.C* をストールさせる（t2）．なお，*thr.2* は *Tx.C* をストールさせている間，*thr.1* へ *Req.A* を定期的に送信し続けることで，A に対するアクセス許可を待つ．その後，*thr.1* の実行が進み，*thr.1* が *Tx.A* のコミットを完了すると，*thr.2* は *Req.A* に対する *Ack* を受信し，A に対するアクセスが可能となったことを知る（t3）．これにより，*thr.2* は *Tx.C* をストール状態から復帰させ，実行を再開する．

しかし，2 つのスレッドが互いの実行するトランザクションの終了を待ち続ける場合，デッドロック状態に陥ってしまう可能性がある．**図 3**(a) の例では，2 つのスレッド *thr.1* および *thr.2* 上でそれぞれトランザクション *Tx.D* および *Tx.E* が実行されており，*thr.1* が `store A` を実行する一方で，*thr.2* が `store B` を実行している．ここで，*thr.1* が `store B` の実行を試みたとすると，B に対するアクセスリクエスト *Req.B* が送信されるが（時刻 t1），この時点で *thr.2* は既に当該アドレスに Write アクセス済みであるため，*thr.1* に *Nack* を返信する．一方，*Nack* を受信した *thr.1* は *thr.2* が *Tx.E* をコミットするまで，自身の *Tx.D* をストールさせる（t2）．その後，*thr.2* が `store A` の実行を試みたとすると（t3），*thr.1* から *Nack* が返信されるため，*thr.2* は *Tx.E* をストールさせる（t4）．このように，2 つのスレッドが互いの実行するトランザクションをストールさせてしまうと，これ以上処理を進めることができなくなり，デッドロック状態に陥ってしまう．

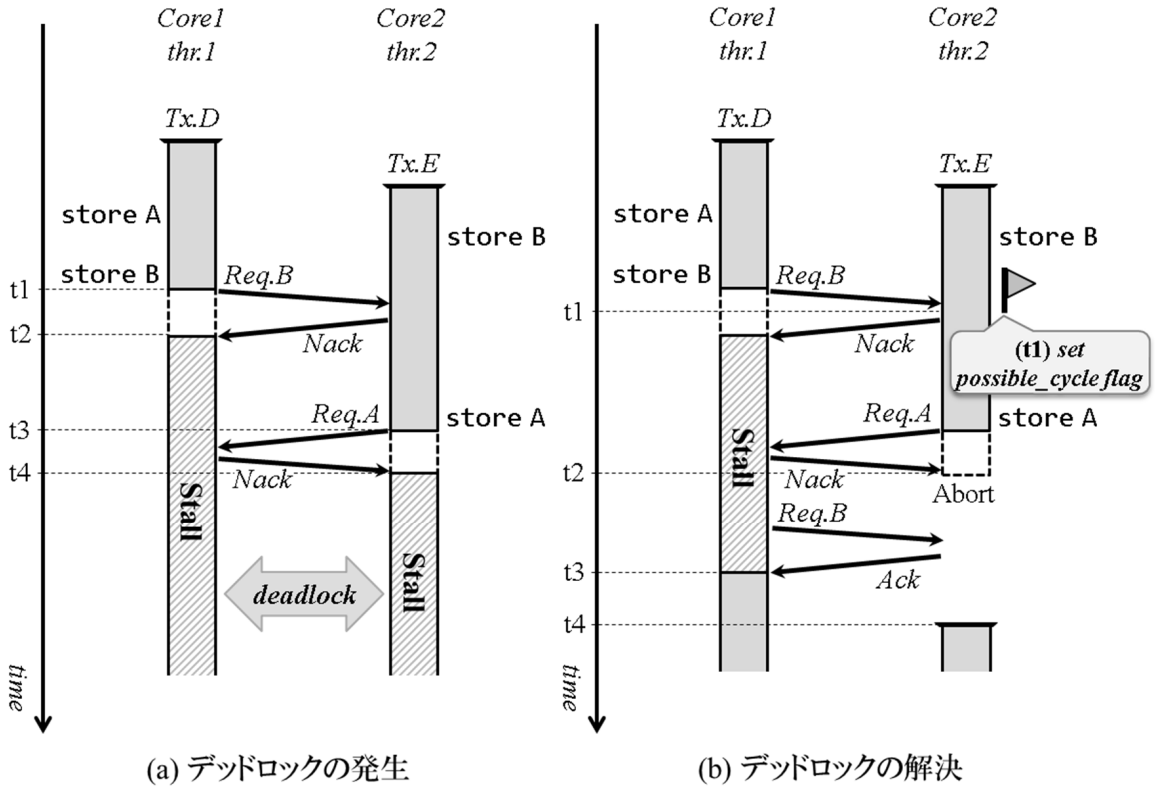


図 3: HTM におけるデッドロックの解決

このようなデッドロック状態を回避するために，一般的な HTM では Transactional Lock Removal[7] の分散タイムスタンプに倣った方法を採用している．この方法では，自身よりも早くトランザクションを開始したスレッドと競合したスレッドが，実行トランザクションをアボートすることでデッドロック状態を回避する．このような動作は，各プロセッサコアに possible_cycle というフラグを追加することで実現されている．ここで，デッドロック状態を possible_cycle フラグにより回避する例を図 3(b) に示す．この例において，*thr.2* は自身より早くトランザクションを開始した *thr.1* に *Nack* を返信する際に possible_cycle フラグをセットする（時刻 t1）．そして，possible_cycle フラグがセットされている状態で，*thr.2* が自身よりも早くトランザクションを開始した *thr.1* から *Nack* を受信した場合，自身の *Tx.E* をアボートする（t2）．一方，*thr.2* が *Tx.E* をアボートしたことにより，*thr.1* は B にアクセスできるようになるため，*Tx.D* の実行を再開できる（t3）．また，*Tx.E* をアボートした *thr.2* は当該トランザクションを再実行する（t4）．このようにして，開始時刻の遅いトランザクションをアボートの対象として選択することでデッドロックを回避しつつ，早く開始されたトランザクションを優先的にコミットさせる．

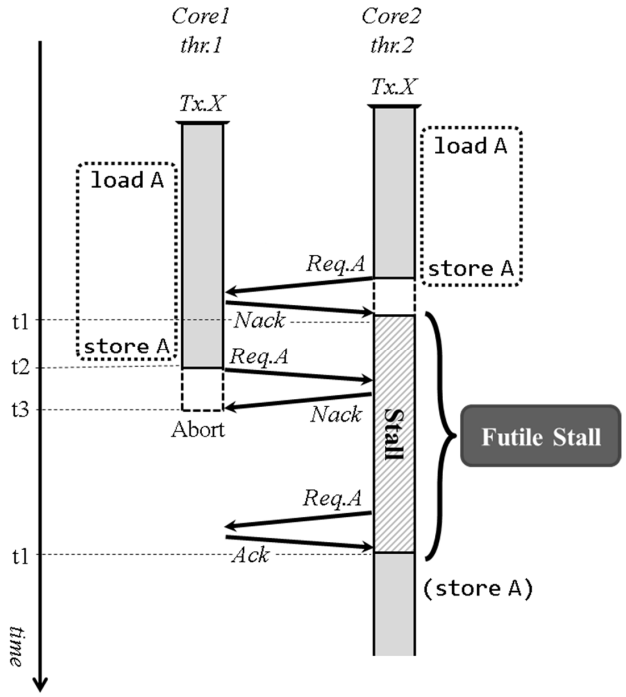


図 4: 共有変数に対する複合操作に起因する問題

2.4　共有変数に対する複合操作に起因する問題

HTM では，前節で述べたような動作により，トランザクションを並列実行した際に発生する競合を検出および解決する．ここで，HTM において並列実行されるトランザクション内には，共有変数に対する Read アクセスの後に Write アクセスをともなう場合が多く見られる．具体的には，複合演算子および複合代入式を用いる複合操作を実現する場合などがこれにあたる．このような同一の共有変数に対する Read，Write の順序でのアクセスを含むトランザクションを複数のスレッドが並列に実行し，それらのスレッドが Read アクセスのみを完了した状態となった場合，その後の Write アクセスによって結果として競合が発生することで，HTM の性能が著しく低下してしまう可能性がある．

図 4 は，共有変数のアドレス A に対する Read，Write の順序でのアクセスを含むトランザクション *Tx.X* を，2 つのスレッド *thr.1* および *thr.2* が並列実行する様子を示している．なお，この図では前節で示した possible_cycle フラグの表記は省略している．まず，両スレッドが `load A` を実行した後に *thr.2* が `store A` の実行を試みる際，*thr.2* は *thr.1* へ A に対するアクセスリクエスト *Req.A* を送信する．この時，*thr.1* が既に A に Read アクセス済みであるため，*thr.1* は競合を検出し，*thr.2* に *Nack* を返信す

表 1: Futile Stall に起因するアボートが全体のアボート数に占める割合

GEMS microbench			
Btree	Contention	Deque	Prioque
39.6%	80.9%	92.9%	99.3%
SPLASH-2			
Barnes	Cholesky	Radiosity	Raytrace
52.2%	92.8%	97.6%	99.4%
STAMP			
Kmeans	Kmeans+	Vacation	Vacation+
87.7%	88.9%	0.7%	0.9%

る．そして，この *Nack* を受信した *thr.2* は *Tx.X* をストールさせる（時刻 t1）．その後，*thr.1* が `store A` の実行を試みる際（t2），*thr.2* が既に A に Read アクセス済であるため，*thr.2* は競合を検出し，*thr.1* に *Nack* を返信する．この時，*thr.1* は自身よりも早くトランザクションを開始した *thr.2* から *Nack* を受信するため，自身の *Tx.X* をアボートすることになる（t3）．このアボートにより，*thr.2* は A にアクセスすることが可能となるため *Tx.X* を再開できるが（t4），その一方でこの間に *thr.1* の処理は一切進行していないことになり，*thr.2* のストールは完全に無駄となってしまう．このように，結果としてアボートされてしまうトランザクションを実行するスレッドと競合することで発生する無駄なストールは **Futile Stall**[8] と呼ばれ，これに起因するアボートとともに HTM のスループットを大きく低下させる要因となりうる．

ここで，上述した Futile Stall とそれに起因するアボートが実際のプログラム上で性能にどの程度の影響を与えているかを調査するために，シミュレーションによる予備評価を行った．なお，ワークロードには HTM の研究で広く用いられている GEMS microbench[9]，SPLASH-2[10] および STAMP[11] を使用し，後述する 6 章に示す環境で評価を行った．この評価の結果を**表 1** に示す．この表には，各プログラムを 16 スレッドで実行した際に発生するアボートの総数のうち，Futile Stall に起因するアボート数の占める割合がそれぞれ示されている．この結果から，多くのプログラムにおけるアボートが Futile Stall に起因するものであり，Futile Stall とそれに起因するアボートを抑制できれば，HTM の大幅な性能向上に繋がることが期待できる．そこで本論文では，この Futile Stall に起因する問題を解決するための 2 つの手法を提案する．

3　関連研究

実行トランザクションをアボートした後にそのトランザクションを途中から再実行することで，再実行に要するコストを抑制する部分ロールバックに関する研究 [12, 13, 14] や，アプリケーションの振る舞いによってバージョン管理や競合検出の方式を動的に変更する研究 [15, 16, 17] など，これまで数多くの HTM に関する研究が行われてきた．しかし，前者の研究では競合しやすいアドレスに対するアクセスの直前から実行が再開されてしまうため，競合がより再発しやすくなるという問題がある．一方で，後者の研究ではバージョン管理と競合検出の方式を変更するために既存のコヒーレンスプロトコルを拡張しているが，この拡張したプロトコルによる動作制御は非常に複雑であり，また実装に必要なハードウェアコストについて詳細に評価していないなどの問題がある．また，これらの方式を変更せずともトランザクションスケジューリングの改良次第で競合の発生を抑制できる可能性があるため，本研究では HTM における効率的なトランザクションスケジューリングに着目する．

このトランザクションスケジューリングについては，これまで主に以下に示す 2 つの方向性から改良手法が提案されてきた．まず，Gaona ら [18] は HTM の消費エネルギーを抑制するという観点から改良手法を提案している．この手法では，複数のトランザクション間で競合が発生した場合に，その競合に関与したトランザクションに実行優先度を設定する．そして，設定した優先度に基づいてトランザクションを逐次実行することで競合を抑制しつつ，HTM における消費エネルギーを削減している．しかし，この手法では既存の HTM に対して最大でも 10%程度の消費エネルギーしか削減されておらず，実行速度も既存の HTM とほぼ同程度の性能に留まっている．

一方で，競合の発生を抑制するという観点から行われた研究として，次の 3 つの手法が挙げられる．まず，Yoo ら [19] は HTM に Adaptive Transaction Scheduling（ATS）と呼ばれるスケジューリング機構を実装し，競合の頻発によって並列性が著しく低下するようなアプリケーションの実行を高速化する手法を提案している．この手法では既存の HTM に対して最大で 97%の実行速度向上を達成しているが，速度向上したプログラムはごく一部であり，その他のプログラムでは目立った速度向上がなされていない．次に，Geoffrey ら [20] は複数のトランザクション内におけるアクセスの局所性を Similarity と定義し，この Similarity がある一定の閾値を超えた場合に，当該トランザクションを逐次的に実行することで競合を抑制する手法を提案している．しかし Geoffrey らの研究では，その性能評価において関連手法のみが比較対象とされている

ため，既存の HTM に対してどの程度速度性能が向上したのかが明確に示されていない．また，Akpinar ら [21] は HTM の性能を低下させるような競合パターンに対する様々な競合解決手法を提案し，既存の HTM に対して最大で 15%の速度向上を達成している．しかし，この研究では HTM においてボトルネックとなりうる競合パターンの発生状況の分析に重点を置いており，これらに対する解決手法はごく単純なものしか提案されていない．

以上に述べた手法は，いずれもアボートや競合の発生回数などの情報のみに基づいてスレッドの振る舞いを決定しており，共有変数にアクセスする順序といった詳細な情報を考慮していない．そのため，HTM の性能を低下させうる競合パターンが根本的には解決されておらず，目立った性能向上を得ることができていない．これに対し本研究では，同一変数のアドレスに対する Read，Write の順序でのアクセスに着目し，このようなアクセスを含むトランザクションに着目したスケジューリングを検討することで，HTM の全体性能の大幅な向上を図る．

なお，Bobba ら [8] は本研究と同様に共有変数に対するアクセス順序に着目し，Store Predictor という機構を用いたスケジューリング手法を提案している．この Store Predictor とは，実行プログラム中で一度でも Read，Write の順序でアクセスされたアドレスを記憶しておくための機構である．Bobba らの手法では，各スレッドがこの機構に記憶されたアドレスに Read アクセスを試みる際に，他のスレッドに対して Read アクセスリクエストではなく，Write アクセスリクエストを送信する．これにより，既に当該アドレスに Read アクセス済みである他のスレッドは WaR アクセスによる競合を検出して *NACK* を返信するため，複数のスレッドが Read アクセスのみを完了した状態となってしまうことを防ぎ，Futile Stall を抑制できる．しかし，この手法で用いる Store Predictor には，実行プログラム中で一度でも Read，Write の順序でアクセスされた変数のアドレスが全て記憶される．そのため，条件分岐などにより必ずしも Read，Write の順序でアクセスされるとは限らないアドレスに対しても，この手法の動作が適用されてしまい，実行するプログラムによっては大幅な性能低下に繋がってしまう可能性がある．そこで本論文では，上述した Bobba らの手法における問題も考慮しつつ，Futile Stall とそれに起因するアボートを抑制する手法を提案する．なお，Bobba らの手法はこれまで述べたように本研究と着眼点が共通しているため，後述する 6 章で提案手法との比較評価を行う．

4　排他的実行による高速化

本章では，同一の共有変数に対する Read，Write の順序でのアクセスを含む処理を排他実行する手法とその具体的な実現方法について述べる．

4.1　同一変数に対する Read，Write アクセスを含む処理の排他実行

2.4 節で述べたように，同一の共有変数に対する Read，Write の順序でのアクセスを含むトランザクションを複数のスレッドが並列実行し，それらのスレッドが Read アクセスのみを完了した状態となってしまうことが，Futile Stall とそれに起因するアボートを発生させる要因となりうる．そこで，本論文では同一変数に対する Read，Write の順序でのアクセスを含む処理を排他的に実行し，複数のスレッドが当該変数に対して Read アクセスのみを完了した状態になることを防ぐことで，Futile Stall およびそれに起因するアボートを抑制する手法を提案する．この手法では，まず Read，Write の順序でアクセスされる共有変数のアドレスを各スレッドが検出する．そして，検出したアドレスに対して Read アクセスしたスレッドは，実行トランザクションをコミットするまで他のスレッドによる当該アドレスへのアクセスを許可しないようにする．

ここで，**図 5** に提案手法を用いた場合の動作例を示す．この例では，共有変数のアドレス A に対する Read，Write の順序でのアクセスを引き起こす複合操作を含むトランザクション *Tx.X* を，3 つのスレッド（*thr.1*〜*thr.3*）が並列実行している．まず，*thr.2* が `load A` を実行し，共有変数に対する複合操作を開始したとする．その後，*thr.1* と *thr.3* が同様に `load A` の実行を試みたとする（時刻 t1，t2）．なお，*thr.1* と *thr.3* のアクセスは RaR アクセスであるため，本来であれば競合として検出されないが，提案手法では *thr.1* と *thr.3* のそれぞれに対して *thr.2* が *Nack* を返信し，これらのスレッドの A に対するアクセスを許可しないようにする．この *Nack* の受信により，*thr.1* と *thr.3* は *Tx.X* をストールさせることになり（t3，t4），これらのスレッドが A に対して Read アクセスを完了した状態となってしまうことを防ぐことができる．そのため，*thr.2* はその後に `store A` の実行を試みる際，*thr.1* および *thr.3* と競合することなく *Tx.X* の処理を進めることが可能となる．これにより，図 4 で示した，Futile Stall およびそれに起因するアボートを抑制できる．その後，*thr.2* が *Tx.X* をコミットしたとすると，A に対するアクセスが可能となる．この例では，*thr.1* の A に対する Read アクセスが先に許可されたと仮定する（t5）．これにより，*thr.1* が A に Read アクセスすることになるため，*thr.1* は *thr.3* から *Req.A* を受信した際に，*thr.2* の場合と同様

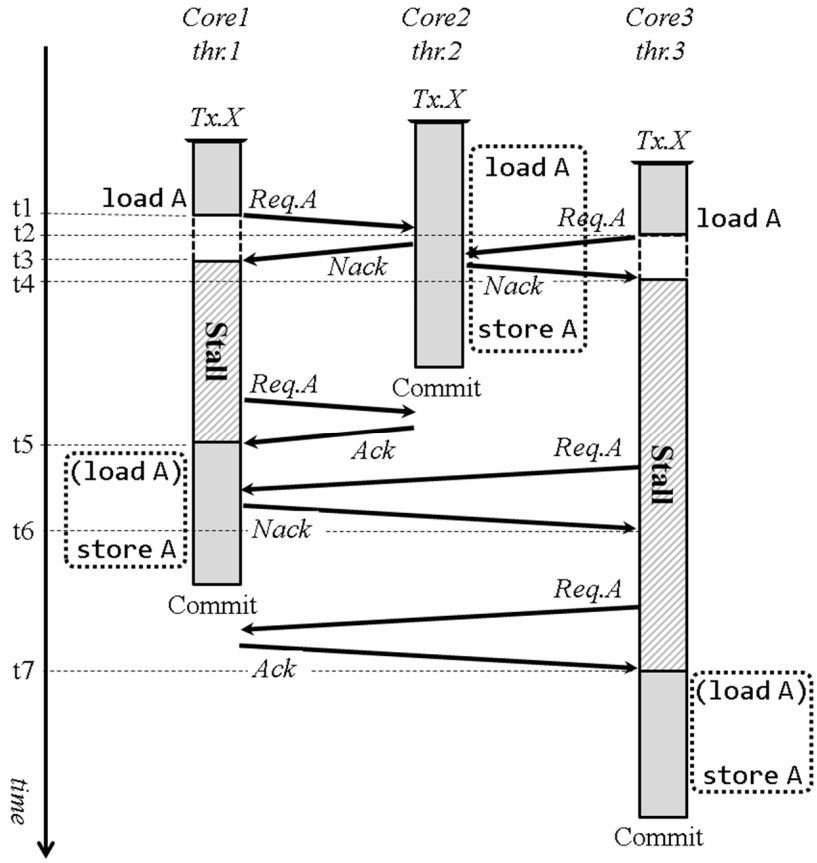


図 5: 同一変数に対する Read，Write アクセスの排他実行

に *Nack* を返信することで *thr.3* のアクセスを許可しないようにする（t6）．そして，*thr.1* が *Tx.X* をコミットした後に，*thr.3* の Read アクセスが許可される（t7）．

4.2　追加ハードウェアと動作モデル

本節では，提案手法を実現するために追加したハードウェアと，その追加ハードウェアを用いた場合の具体的な動作モデルについて述べる．

4.2.1　拡張ハードウェア構成

前節で述べた提案手法の動作を実現するにあたり，Read，Write の順序でアクセスされる共有変数へのアクセスを許可するか否かを，各スレッドが判定する必要がある．そこで，各コアの L1 キャッシュラインに以下の 1bit のフィールドを追加する．

Compound-Operation bit(C-bit):

当該ライン上のアドレスが Read，Write の順序でアクセスされたか否かを示すビット

各スレッドは，自身が共有変数のアドレスに対してRead，Writeの順序でアクセスしたと判断した場合に，当該アドレスに対応するCビットをセットする．そして，他のスレッドからアクセスリクエストを受信したスレッドは，アクセス対象のアドレスに対応するCビットおよびRビットを参照する．これにより，リクエストを送信したスレッドに対して*Nack*を返信すべきか否かが判定される．

4.2.2 Cビットのセット方法

本項では，Read，Writeの順序でアクセスされる共有変数のアドレスを各スレッドが検出し，そのアドレスに対応するCビットをセットするまでの動作について，**図6**の動作例を用いて説明する．この例では，共有変数のアドレスAに対するRead，Writeの順序でのアクセスを含むトランザクション*Tx.X*を，3つのスレッド（*thr.1*〜*thr.3*）が並列実行している．なおこの例では，各コアのL1キャッシュにおけるタグフィールドおよびインデクスフィールドを，addrというフィールドとして簡略化して示しており，このaddrフィールドにはアドレスが格納されているものとする．

この例において，まず各スレッドが`load A`を実行した後，*thr.2*が`store A`の実行を試み，*thr.1*および*thr.3*にアクセスリクエスト*Req.A*を送信したとすると（時刻t1），これらのスレッドは競合を検出する．この競合により，*thr.1*および*thr.3*から*Nack*が返信されるため，*thr.2*は*Tx.X*をストールさせる（t2）．続いて，*thr.1*と*thr.3*が`store A`の実行を試みる際，*thr.2*との間でそれぞれ競合が検出され，*thr.2*から*thr.1*と*thr.3*に対して*Nack*が返信される．この*Nack*により，*thr.1*と*thr.3*が*Tx.X*をストールさせると，これらのスレッド間でデッドロックが発生してしまうため，*thr.1*および*thr.3*は*Tx.X*をアボートする．この時，AがRead，Writeの順序でアクセスされたか否かをチェックするために，*thr.1*と*thr.3*はAに対応するRビットを参照する．図6の例では，*Core.1*および*Core.3*におけるL1キャッシュ上のAに対応するRビットが既にセット済みであることから，*thr.1*および*thr.3*はAがRead，Writeの順序でアクセスされるアドレスであることが分かる．そのため，*thr.1*と*thr.3*はAに対応するCビットをセットする（t3，t4）．その後，トランザクションをアボートした*thr.1*と*thr.3*はAがRead，Writeの順序でアクセスされるアドレスであることを*Info.*メッセージにより*thr.2*に伝え（t5，t6），*thr.2*もAに対応するCビットをセットした状態となる．なお，この*Info.*メッセージはコヒーレンスプロトコルを拡張することで新たに定義する．

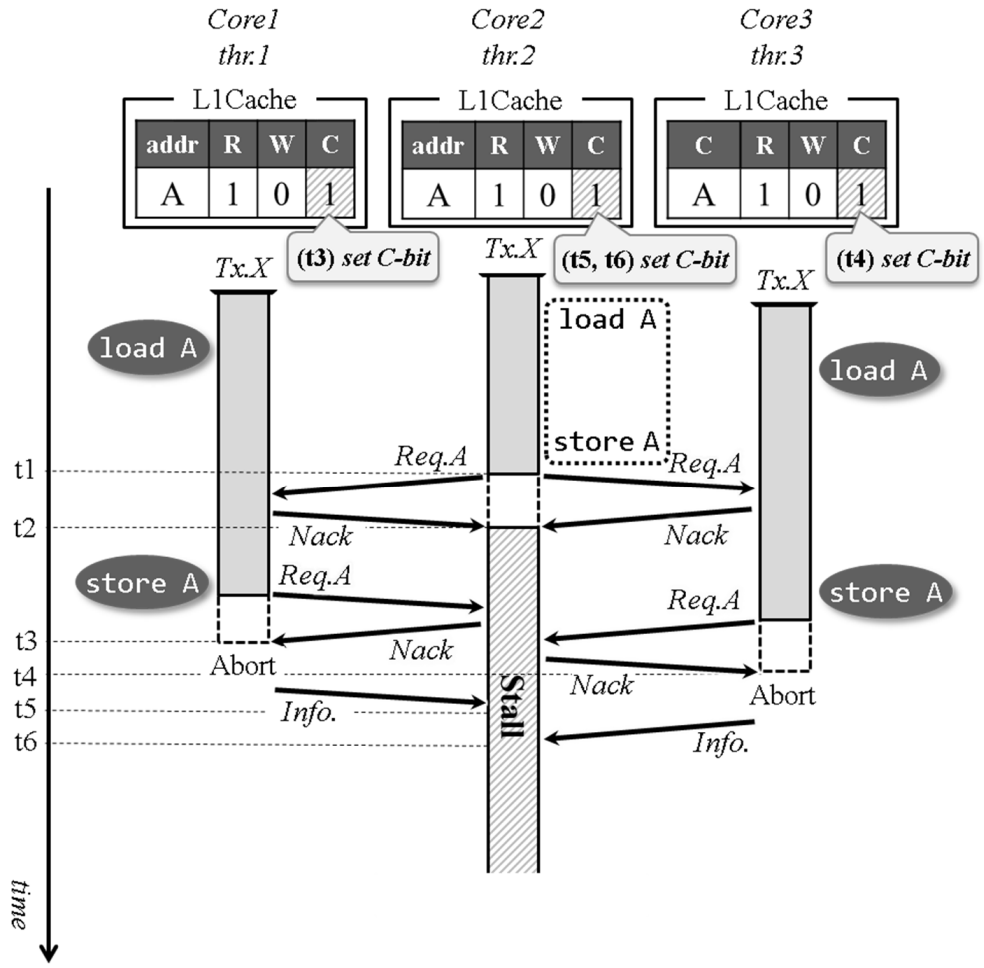


図 6: C ビットをセットする動作

4.2.3 C ビットの利用による排他的実行の実現

前項で述べた動作によりセットされた C ビットを利用することで，同一変数に対する Read，Write の順序でのアクセスを排他的に実行する動作例を**図 7** および**図 8** に示す．これらの例では，共有変数のアドレス A に対する Read，Write の順序でのアクセスを含むトランザクション *Tx.X* を，3 つのスレッド（*thr.1*〜*thr.3*）が並列実行しており，さらに各コアの L1 キャッシュライン上の A に対応する C ビットが既にセットされているとする．

まず，図 7 に示すように，*thr.2* が C ビットのセットされている A に対して `load A` を実行したとする．続いて，*thr.1* が `load A` の実行を試みたとすると，*thr.1* は *thr.2* に A に対するアクセスリクエスト *Req.A* を送信する．この *Req.A* を受信した *thr.2* は，*thr.1* の Read アクセスを許可するか否かを判定するために，A に対応する C ビットと R

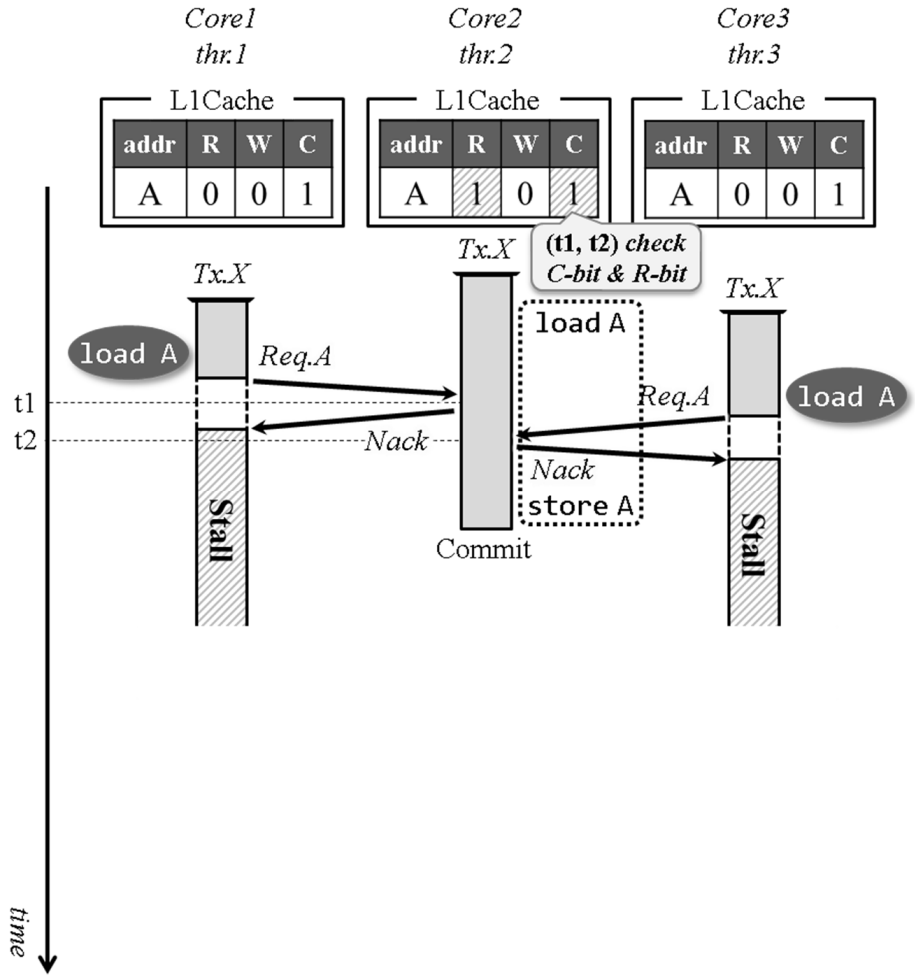


図 7: 排他的実行の実現

ビットを参照する（時刻 t1）．この時，どちらのビットもセット済みであるなら，*thr.2* は *thr.1* の Read アクセスを許可すべきではないと判断する．そのため，*thr.2* は *thr.1* に *Nack* を返信し，Read アクセスを許可しないようにする．その後，*thr.3* が `load A` の実行を試み，*thr.2* に *Req.A* を送信する際も，この *Req.A* を受信した *thr.2* は *thr.3* に *Nack* を返信する．これにより，*thr.2* は無駄なストールをすることなく *Tx.X* をコミットできる．

ここで，*thr.2* が *Tx.X* をコミットした後の動作例を図 8 に示す．この例では，*thr.2* により *thr.1* の Read アクセスが先に許可されたと仮定する．そのため，*thr.1* は A に Read アクセス済みの状態となる（時刻 t3）．この状態で，*thr.1* が *thr.3* から *Req.A* を受信した場合，*thr.1* は *thr.3* の Read アクセスを許可するか否かを判定するために，A に対応する C ビットと R ビットを参照する（t4）．この時，どちらのビットもセット

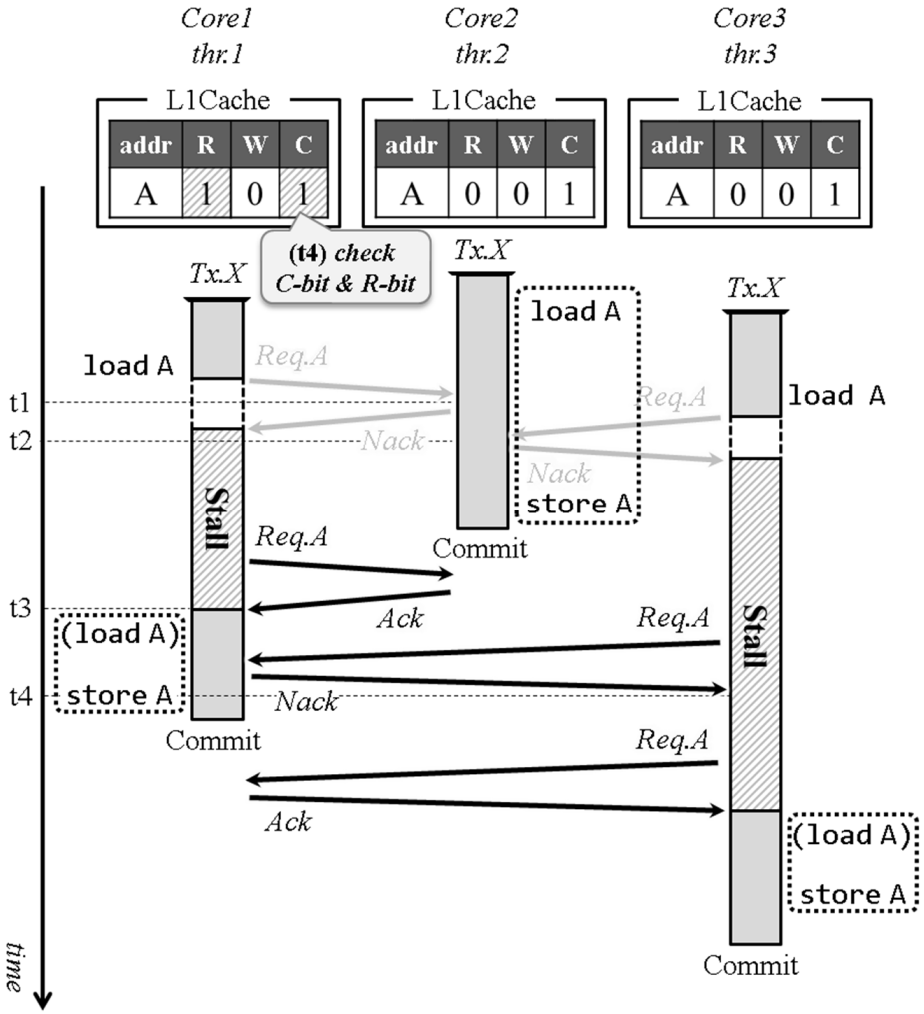


図 8: アクセスを許可されたスレッドの動作

済みであるなら，*thr.1* は *thr.3* に *Nack* を返信し，*thr.3* の Read アクセスを許可しないようにする．これにより，*thr.1* は無駄なストールをすることなく *Tx.X* をコミットでき，その後にアクセスが許可される *thr.3* も同様に *Tx.X* をコミットできる．以上のように，追加した C ビットを利用することで，同一変数に対する Read，Write の順序でのアクセスを排他的に実行する．

4.2.4 C ビットのクリア方法

ここで，追加した C ビットをクリアする方法について述べる．4.2.3 項でも述べたように，この C ビットは対応するキャッシュライン上のアドレスが Read，Write の順序でアクセスされたことを示すものであり，同一変数に対する Read，Write の順序でのアクセスを排他実行するために用いられる．しかし，実行するプログラム内容によっ

ては，プログラム中のあるトランザクション内でRead，Writeの順序でアクセスされるようなアドレスが，別のトランザクション内ではReadアクセスしかされない場合もある．そのため，このCビットがセットされているアドレスに対して，Readアクセスしか行わないようなトランザクションが実行された場合，本来ならば並列実行できるはずのトランザクションが逐次実行されてしまい，結果として性能低下を引き起こしてしまう可能性がある．そこで，各スレッドは実行トランザクションをコミットした際に，Cビットのセットされているアドレスが Writeアクセスされたか否かを Wビットを参照することによりチェックする．このWビットがセットされていなかった場合，各スレッドは自身が当該アドレスに対してReadアクセスのみしか行わなかったと判断してCビットをクリアする．

具体的な動作例を**図9**に示す．この例では，共有変数のアドレスAに対してReadアクセスのみしか行わないトランザクション *Tx.Y* を，3つのスレッド（*thr.1*〜*thr.3*）が並列実行しており，さらに各スレッドが既にAに対応するCビットをセット済みであるとする．この例において，まず *thr.2* が `load A` を実行し，AにReadアクセス済みの状態となる．その後，*thr.1* と *thr.3* が同様に `load A` の実行を試みた際，*thr.2* は提案手法の動作に従って，これらのスレッドのAに対するReadアクセスを許可しない．そのため，本来であれば並列実行できたはずの処理部分が逐次実行されるとともに，多くのスレッドが実行トランザクションをストールさせてしまうことで，並列度が著しく低下する．このような動作をこれ以降引き起こさないようにするために，*Tx.Y* のコミット時に，各スレッドはCビットのセットされているAに対応するWビットを参照する．このWビットがセットされていなかった場合，この *Tx.Y* 内でAに対するWriteアクセスがなされなかった，つまりAに対するReadアクセスしかなされなかったと各スレッドは判断し，Aに対応するCビットをクリアする．以上のように動作させることで，これ以降にReadアクセスしか行わないトランザクションが逐次実行されてしまうことを防ぐ．

5　投機的実行による高速化

本章では，トランザクション内で同一の共有変数に対するRead，Writeの順序でのアクセスを完了した時点で，当該トランザクションのコミットに先立って他スレッドによる当該変数への投機的アクセスを許可する手法とその実現方法について述べる．

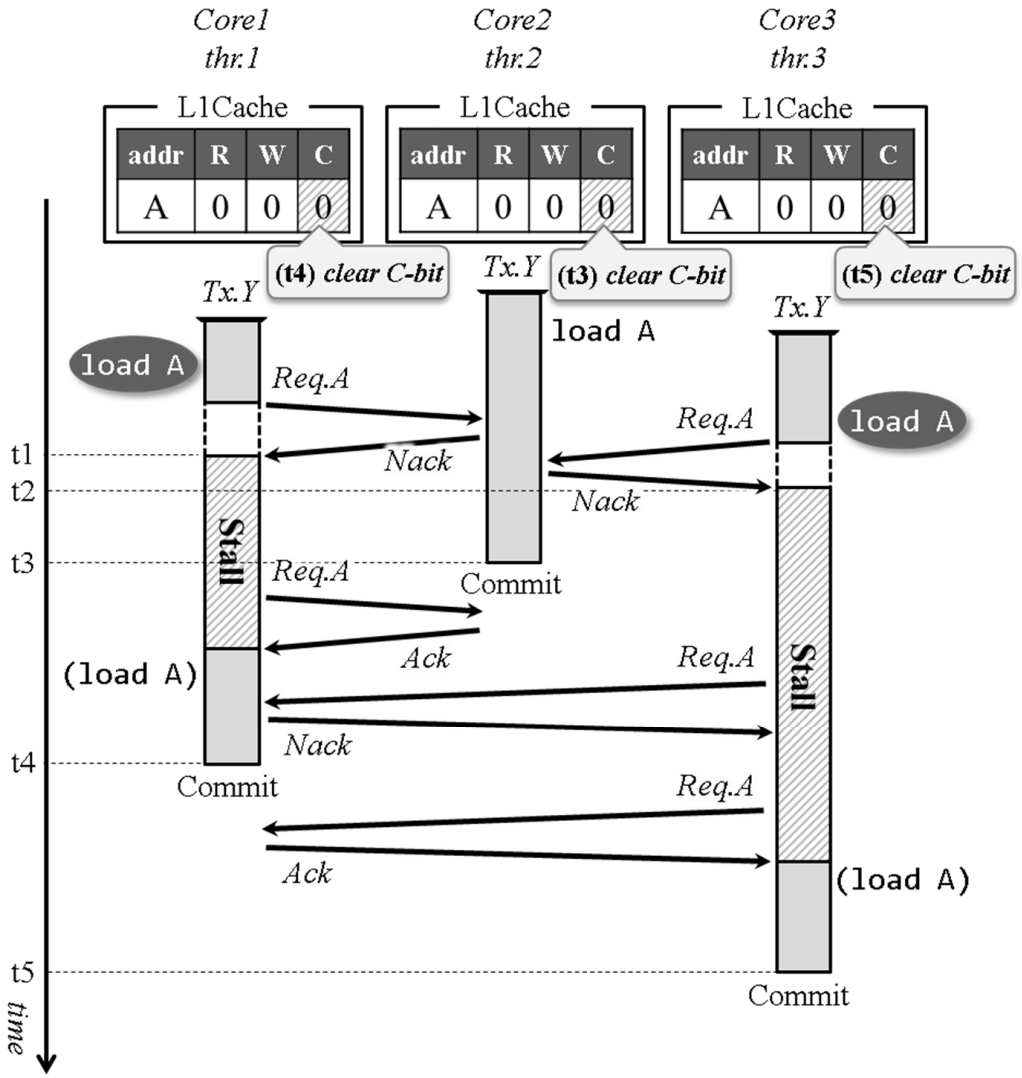


図 9: C ビットをクリアする動作

5.1 投機的アクセスの許可による高速化

4 章で述べた手法では，同一の共有変数に対する Read，Write の順序でのアクセスを含むトランザクションを実行したスレッドが，当該トランザクションをコミットするまで他のスレッドによる当該変数へのアクセスを許可しないことで排他的な実行を実現した．ここで，一般的なプログラムで実行されるトランザクションには，同一の共有変数に対する Read，Write の順序でのアクセスの後に，当該変数に対する再度のアクセスを含まないものがある．このようなトランザクションを複数のスレッドが並列実行する場合，あるスレッドが同一の共有変数に対する Read，Write の順序でのアクセスを完了した時点で，他スレッドが当該変数に投機的にアクセスしたとしても，結

果として一貫性が保たれる可能性がある．そのため，同一の共有変数に対する Read, Write の順序でのアクセスを含む処理を排他的に実行する手法にはさらなる高速化の余地があると考えられる．そこで本論文では，トランザクション内で同一の共有変数に対する Read，Write の順序でのアクセスが完了した時点で，当該トランザクションのコミットに先立って他スレッドによる投機的アクセスを許可する手法を併せて提案する．

この手法では，まず Read，Write の順序でアクセスされるアドレスを各スレッドが検出するとともに，実行トランザクション内で同一変数に対する Read，Write の順序でのアクセスが完了する地点を各スレッドが記憶する．そして，検出したアドレスに対して Read アクセスしたスレッドは，自身が当該アドレスに対する Write アクセスを完了するまで，他のスレッドによる当該アドレスへの投機的アクセスを *Wait* リクエストという通知を送信することで待機させる．なお，この *Wait* リクエストはコヒーレンスプロトコルを拡張することで新たに定義する．その後，Write アクセスを完了した時点で，待機スレッドに対して実行の再開を通知する *Ack* を送信することで，自身のコミットに先立って待機スレッドの投機的アクセスを許可する．ここで，このように投機的アクセスが許可された場合は，当該アドレスに対する Read，Write の順序でのアクセスを完了したスレッドにより更新された値を用いて，投機的アクセスを許可されたスレッドが実行トランザクションの処理を進めることになる．そのため，投機的アクセスが許可されたスレッドよりも先に，当該アドレスに対する Read，Write の順序でのアクセスを完了したスレッドが実行トランザクションをコミットしなければならないという，コミット順序に関する制約が発生する．このコミット順序を制御するために，Read，Write の順序でのアクセスを完了したスレッドは実行トランザクションをコミットした際，投機的アクセスを許可したスレッドに対して *Committed* メッセージという通知を送信する．なお，この *Committed* メッセージは *Wait* リクエストと同様にコヒーレンスプロトコルを拡張することで新たに定義する．

ここで**図 10** に，上述した投機的アクセスを許可する手法を用いた場合の動作例を示す．この例では，共有変数のアドレス A に対する Read，Write の順序でのアクセスの後に，当該変数に再度アクセスしないようなトランザクション *Tx.X* を，2 つのスレッド *thr.1* および *thr.2* が並列実行している．まず，*thr.2* が `load A` を実行し，A に対して Read アクセス済みとなった後，*thr.1* が `load A` の実行を試みたとする（時刻 t1）．この時，*thr.1* は *thr.2* へ A に対するアクセスリクエスト *Req.A* を送信するが，この時点で A は *thr.2* により Read アクセス済みであるため，*thr.2* は *thr.1* に対して *Wait* リ

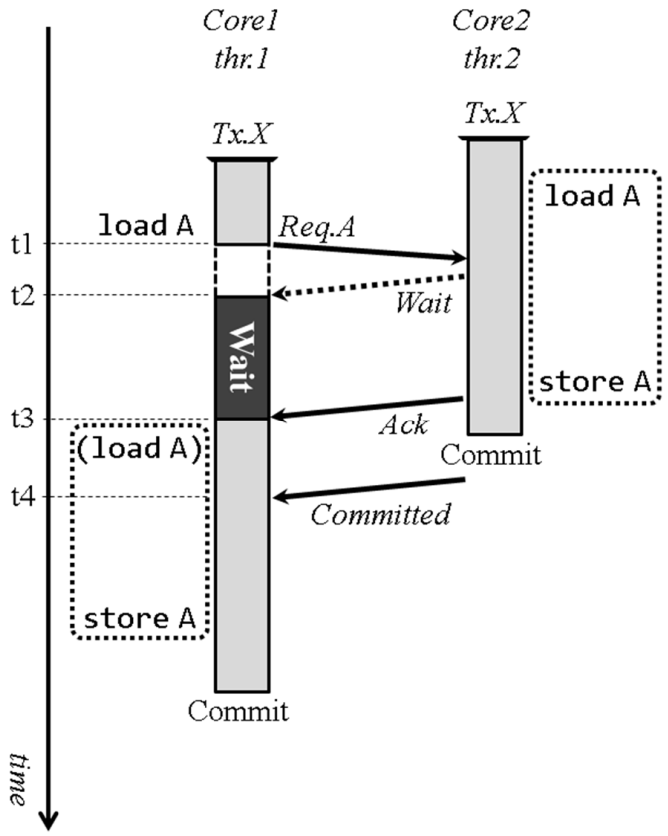


図 10: 投機的な実行による高速化

クエストを送信し，*thr.1* のアクセスを待機させる（t2）．その後，*thr.2* が `store A` を実行したとすると，*thr.2* はこれ以降に *Tx.X* 内で A にアクセスしないため，A に対する投機的アクセスを許可することが可能となり，*thr.2* は待機スレッドである *thr.1* に対して *Ack* を送信し，*thr.1* のアクセスを許可する（t3）．これにより，*thr.2* のコミットに先行して，*thr.1* が A に投機的にアクセスできる．なお，このように *thr.2* が *thr.1* の投機的アクセスを許可したことで，*thr.2* の実行する *Tx.X* 内で更新された値を用いて *thr.1* が *Tx.X* の処理を進めることになる．そのため，*thr.2* は *thr.1* よりも先に *Tx.X* をコミットし，トランザクションの Serializability を保証する必要がある．したがって，*thr.2* は *Tx.X* をコミットした際，投機的アクセスを許可した *thr.1* に対して *Committed* メッセージを送信し，自身が *Tx.X* をコミットしたことを伝えるようにする（t4）．以上のように動作することで，Futile Stall およびそれに起因するアボートを抑制するとともに，投機的アクセスの許可により 4 章で述べた手法のさらなる高速化を期待できる．

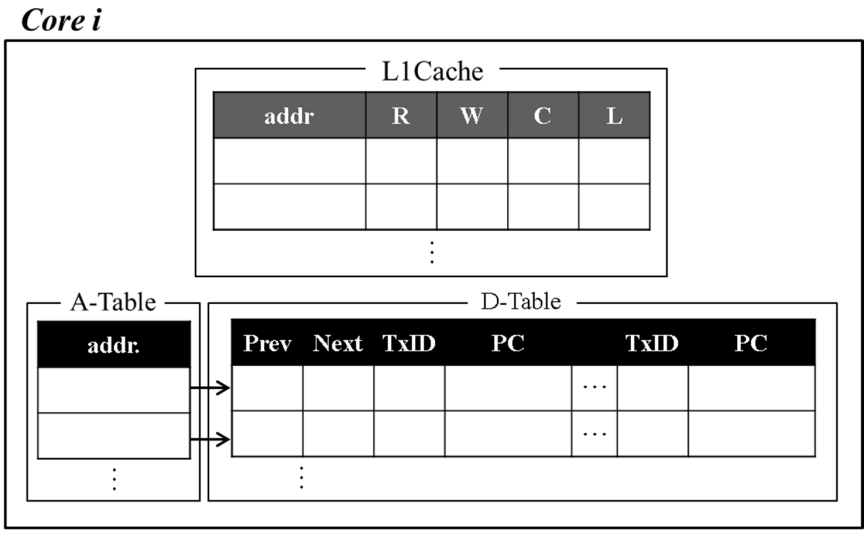


図 11: 拡張したハードウェア構成

5.2 追加ハードウェアと動作モデル

本節では，投機的アクセスを許可する手法を実現するために追加したハードウェアと，その追加ハードウェアを用いた場合の具体的な動作モデルについて述べる．

5.2.1 拡張ハードウェア構成

拡張したハードウェア構成を**図 11** に示す．なお図 11 に示す L1 キャッシュでは，タグフィールドおよびインデクスフィールドを，addr というフィールドとして簡略化して示しており，この addr フィールドにはアドレスが格納されるものとする．ここで，この手法を実現するにあたり，まず他のスレッドによるアクセスを待機させるか否かを各スレッドが判定する必要がある．そこで，各コアの L1 キャッシュラインに 4 章で示した C ビットに加えて，以下に示す 1bit のフィールドを追加する．

Lock bit(L-bit):

当該ライン上のアドレスに対する対応する共有変数に対する，Read，Write の順序でのアクセスを含むトランザクションにおいて，Read アクセスが完了済みかつ Write アクセスが未完了であることを示すビット．すなわち，Read アクセスから Write アクセスまでの処理区間を排他的に実行中であるか否かを示す．

またこの手法では，トランザクション内で Read，Write の順序でアクセスされる共有変数に対して，自身が Write アクセスを完了したか否かを各スレッドが判断する．この判断のために，Read，Write の順序でアクセスされる変数のアドレス，そのようなアクセスを含むトランザクションの ID，そして Write アクセスが実行される時点における

プログラムカウンタの値を記憶する機構が必要となる．さらに，待機スレッドに対するアクセス許可やトランザクションのコミット順序の制御を行うために，各スレッドが動作するコアの番号を記憶する機構が必要となる．そこで，それらの情報を記憶するための2つの表を各コアに追加する．この2つの表をそれぞれ **Address Table(A-Table)**，**Dependence Table(D-Table)** と呼ぶ．なお，これらの表は以下に示すフィールドで構成される．

Address Table(A-Table)

Target-Address(addr.):

Read，Write の順序でアクセスされる共有変数のアドレス

Dependence Table(D-Table)

Prev-Core(Prev):

Address Table に記憶したアドレスに対する投機的アクセスを自身に許可したコアの番号

Next-Core(Next):

Address Table に記憶したアドレスに対する投機的アクセスを自身が許可すべきコアの番号

Target-TxID(TxID):

Address Table に記憶したアドレスに対する Read，Write の順序でのアクセスを含むトランザクションの ID

Target-PC(PC):

Address Table に記憶したアドレスに対する Read，Write の順序でのアクセスを含むトランザクション内で，Write アクセスが実行される時点における PC の値

各スレッドは，C ビットおよび L ビットが共にセット済みである場合に，対応するアドレスに対する他のスレッドによるアクセスを待機させる．また，Target-Address，Target-TxID および Target-PC に格納された情報に基づいて，Read，Write の順序でアクセスされる共有変数に対する Write アクセスを完了したか否かを各スレッドが判断する．そして，Prev-Core および Next-Core に格納された情報に基づいて，待機スレッドに対するアクセス許可とコミット順序の制御を各スレッドが行う．なお実行するプログラムによっては，ある共有変数が複数のトランザクション内で Read，Write の順序でアクセスされる可能性がある．そのため，図 11 に示すように，Dependence Table はそれぞれの Target-Address に対して複数の Target-TxID および Target-PC を

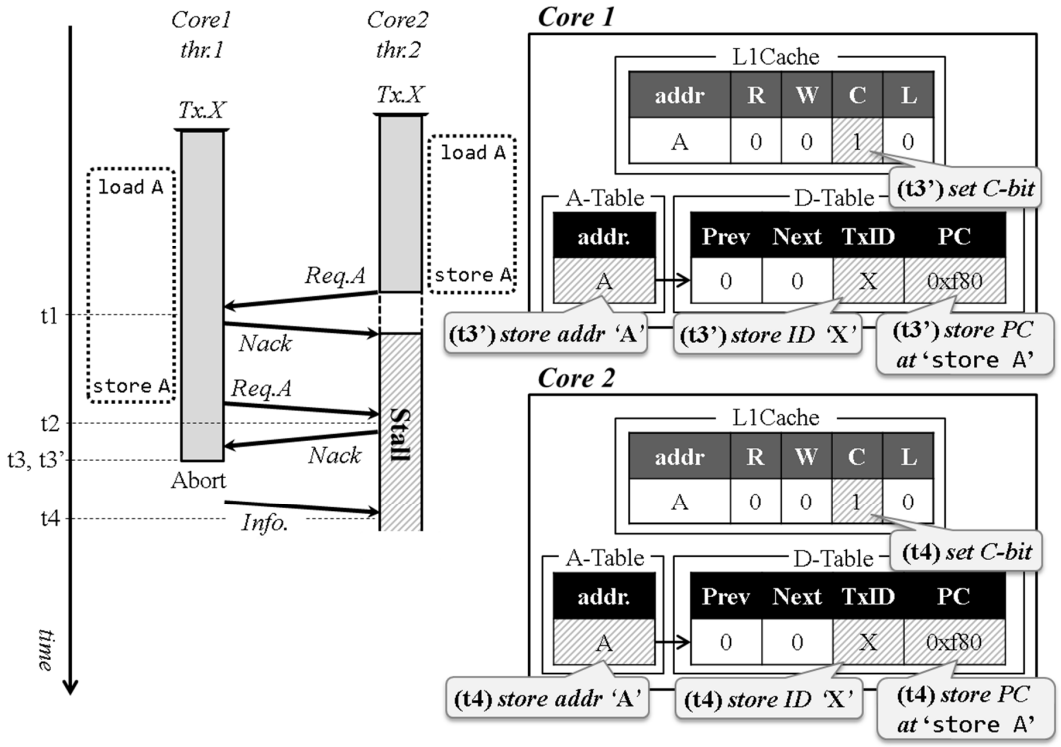


図 12: Read，Write の順序でアクセスされるアドレスを記憶する動作

記憶できるように構成されている．

5.2.2 Read，Write の順序でアクセスされるアドレスの記憶

本項では，Read，Write の順序でアクセスされるアドレスに対応する C ビットがセットされるとともに，そのアドレスが Address Table に記憶されるまでの動作について，**図 12** の動作例を用いて説明する．この例では，共有変数のアドレス A に対する Read，Write の順序でのアクセスを含むトランザクション *Tx.X* を，2 つのスレッド *thr.1* および *thr.2* が並列実行している．また，Dependence Table に記憶される Target-TxID および Target-PC は 1 組であると仮定している．

この例において，まず各スレッドが `load A` を実行した後，*thr.2* が `store A` の実行を試みたとすると，*thr.1* はこの時点で既に A に対して Read アクセス済みであることから競合を検出する（時刻 t1）．この競合により，*thr.1* から *Nack* が返信されるため，*thr.2* は *Tx.X* をストールさせる．続いて，*thr.1* が `store A` の実行を試み，A に対するアクセスリクエスト *Req.A* を送信したとすると，*thr.2* は競合を検出し，*thr.1* に *Nack* を返信する（t2）．ここで，これらのスレッド間でデッドロックが発生してしまうため，*thr.1* が *Tx.X* をアボートする（t3）．この時，A が Read，Write の順序で

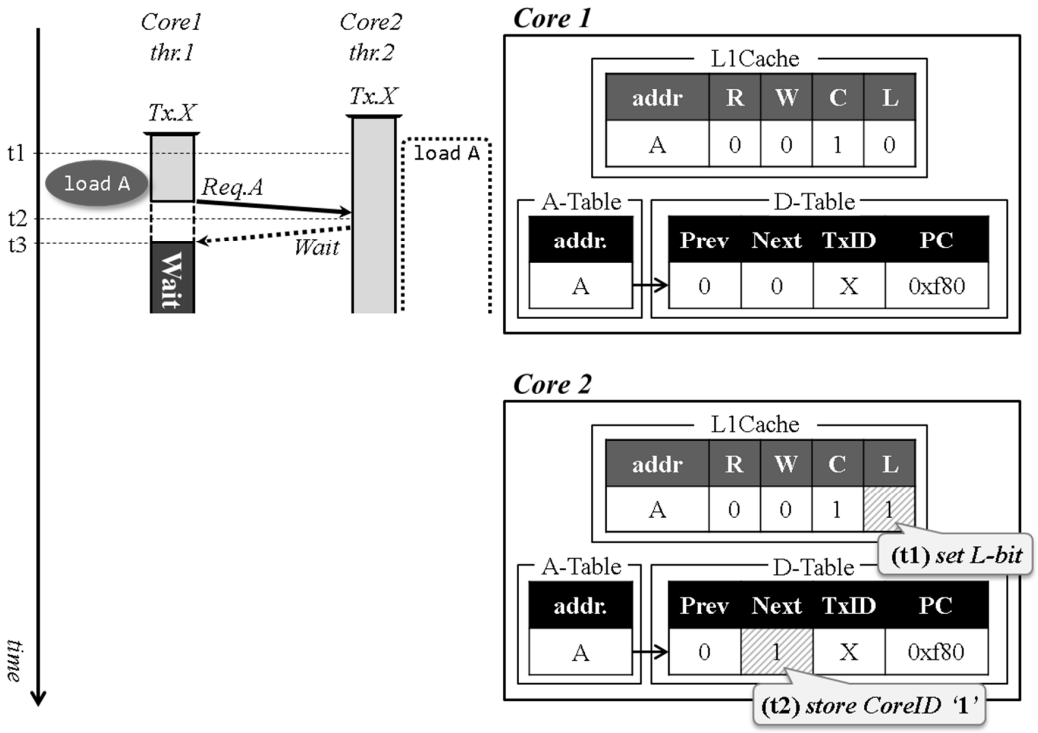


図 13: *Wait* リクエストによってアクセスを待機させる動作

アクセスされたか否かをチェックするために，*thr.1* は A に対応する R ビットを参照する．図 12 の例では，*Core.1* における L1 キャッシュ上の A に対応する R ビットが既にセット済みであることから，*thr.1* は A が Read，Write の順序でアクセスされるアドレスであることが分かる．そのため，*thr.1* は A に対応する C ビットをセットするとともに，アドレス A を Address Table に格納する（t3’）．さらに，この時 *thr.1* は自身が試みた `store A` の実行地点を，A に対する Read，Write の順序でのアクセスが完了する地点であると判断し，この時点におけるプログラムカウンタの値および現在実行しているトランザクションの ID である ’X’ を Dependence Table に格納する．これにより，以後 *thr.1* は再度 *Tx.X* を実行した際に，A に対する Write アクセスの実行を完了したか否かを判断できる．その後，*thr.1* は追加ハードウェアに記憶した A に関する情報を，*Info.* メッセージによって *thr.2* に伝える（t4）．なお，この *Info.* メッセージはコヒーレンスプロトコルを拡張することで新たに定義する．

5.2.3 依存関係情報の利用による投機的な実行の実現

5.2.2 項で述べた動作により記憶した情報を利用し，Read，Write の順序でアクセスされるアドレスに対する投機的なアクセスを許可する動作例を**図 13**，**図 14** および**図**

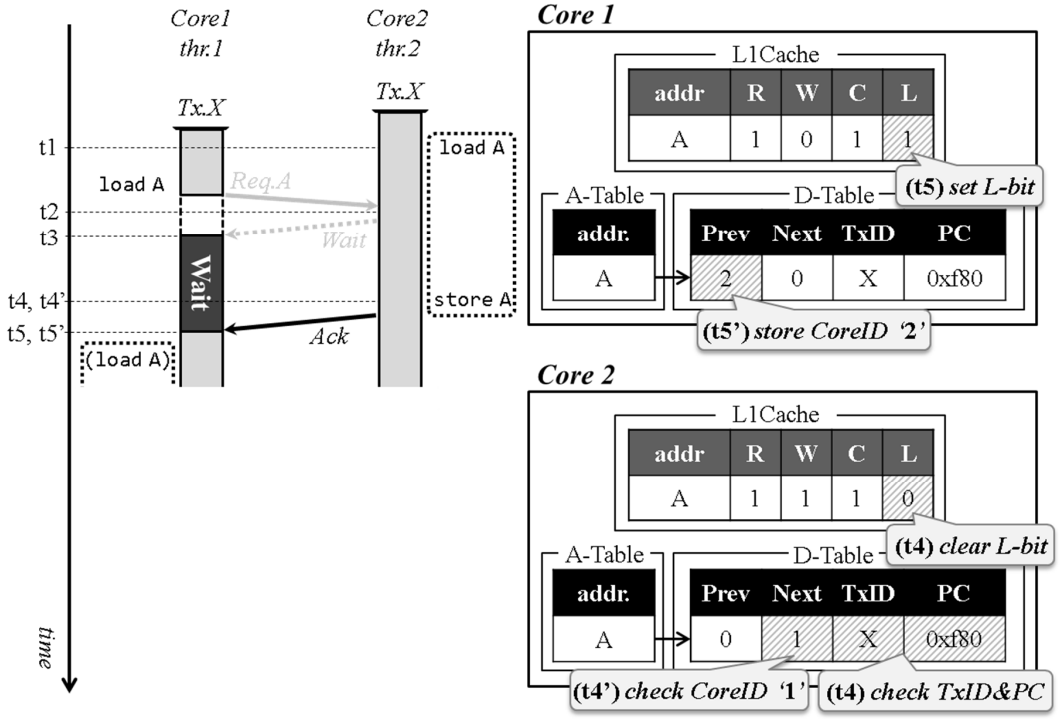


図 14: 待機スレッドの投機的アクセスを許可する動作

15 に示す．これらの例では，共有変数のアドレス A に対する Read，Write の順序でのアクセスを含むトランザクション *Tx.X* を，2 つのスレッド *thr.1* および *thr.2* が並列実行している．また，各コアの L1 キャッシュ上の A に対応する C ビットが既にセットされており，さらに 2 つの表にアドレス 'A'，トランザクション ID 'X' およびプログラムカウンタの値 '0xf80' が格納されているとする．

まず，図 13 に示すように，*thr.2* が `load A` の実行を試みたとする．この時，C ビットのセットされている A に Read アクセスした *thr.2* は，自身が A に対する Read，Write の順序でのアクセスを含む処理のうち，その Read アクセスから Write アクセスまでの処理区間を排他的に実行中であると判断し，A に対応する L ビットをセットする（時刻 t1）．その後，*thr.1* が同様に `load A` の実行を試み，*thr.2* へ A に対するアクセスリクエスト *Req.A* を送信したとする．この *Req.A* を受信した *thr.2* は，A に対応する C ビットと L ビットを参照する．この時，これらのビットがセット済みであることから，*thr.2* は *thr.1* のアクセスを待機させる必要があると判断する．そのため，*thr.2* は *thr.1* に *Wait* リクエストを送信し，*thr.1* の A に対する Read アクセスを待機させる（t2）．また，同時に *thr.2* は，*thr.1* のアクセスを後に許可できるようにするために，*thr.1* が

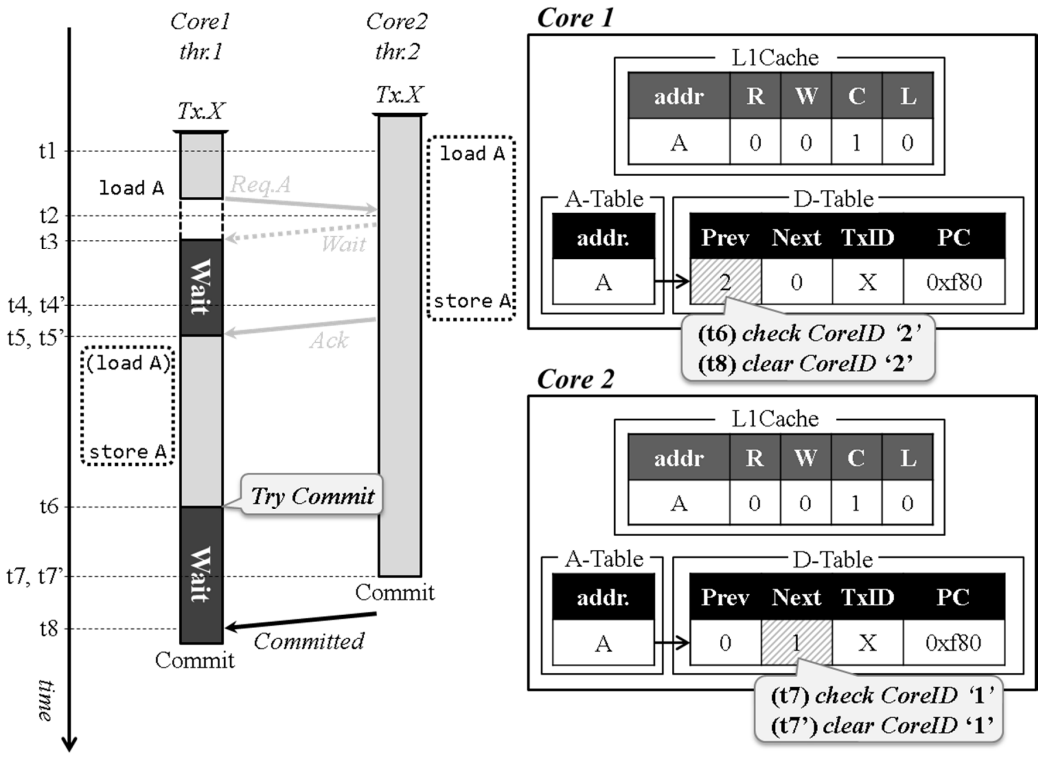


図 15: トランザクションのコミット順序を制御する動作

動作しているコア番号を取得し，この値である 1 を Dependence Table の Next-Core に格納する．

ここで，この例において，*thr.2* が C ビットおよび L ビットが共にセット済みである A に `store A` を試みたとする（図 14，時刻 t4）．この時，*thr.2* は Read，Write の順序でアクセスされる A に対する Write アクセスを完了したか否かを確認するために，Dependence Table を参照する．そして，*thr.2* は自身が実行しているトランザクションの ID およびこの時点におけるプログラムカウンタの値を，Dependence Table の Target-TxID および Target-PC に格納されている値と比較する．この時，これらの値が一致したとすると，*thr.2* は A に対する Write アクセスを完了したと判断し，A に対応する L ビットをクリアする．これにより，*thr.1* が待機している A に対する投機的アクセスを，*thr.2* が許可できるようになるため，*thr.2* は Dependence Table の Next-Core に格納されているコア番号を参照する（t4'）．その結果，*thr.2* はコア番号 1 を取得するため，*Core.1* において Read アクセスを待機している *thr.1* に *Ack* を送信する．この *Ack* を受信した *thr.1* は，A に対する Read アクセスが *thr.2* により許可されたと判断し，待機していた Read アクセスを投機的に実行する（t5）．なお，このように *thr.2*

が *thr.1* の投機的アクセスを許可したことで，*thr.2* の実行する *Tx.X* 内で更新された値を用いて *thr.1* が *Tx.X* の処理を進めることになる．そのため，*thr.2* は *thr.1* よりも先に *Tx.X* をコミットしなければならないという，トランザクションのコミット順序に関する制約が発生する．このコミット順序を制御するために，*thr.1* は *thr.2* が動作しているコア番号を取得し，この値である 2 を Dependence Table の Prev-Core に格納する（t5'）．

その後，各スレッドの処理が進み，*thr.1* が *Tx.X* のコミットに到達した際（図 15，時刻 t6），*thr.1* は Dependence Table の Prev-Core に格納されているコア番号を参照する．その結果，コア番号 2 が取得されるため，*Core.2* 上で動作する *thr.2* が実行トランザクションをコミットするまで，*thr.1* は *Tx.X* のコミットを待機する．その後，*thr.2* が *Tx.X* のコミットした際（t7），*thr.2* は Dependence Table の Next-Core に格納されているコア番号を参照する．その結果，1 というコア番号が取得されるため，*thr.2* は *Tx.X* をコミットしたことを *Core.1* に伝える必要があると判断し，*Core.1* に *Committed* メッセージを送信するとともに，Dependence Table の Next-Core に格納されているコア番号をクリアする（t7'）．この *Committed* メッセージを受信した *thr.1* は，*thr.2* の実行する *Tx.X* がコミットされたと判断し，Dependence Table の Prev-Core に格納されているコア番号をクリアするとともに，自身の *Tx.X* をコミットする（t8）．

5.3　トランザクション定義を満たさない場合の例外処理

これまで述べたように，提案手法では共有変数に対する Read，Write の順序でのアクセスを完了した時点で，他のスレッドによる投機的アクセスを許可することで高速化を図っている．しかし，この手法では投機的アクセスの許可により，メモリ一貫性が欠如した状態やデッドロック状態に陥ってしまう可能性がある．そこで，本節ではそのような問題を回避するために設定した，2 つの例外処理についてそれぞれ説明する．

まず，1 つめの問題であるメモリ一貫性の欠如した状態に陥ってしまう例を**図 16** に示す．この例では，共有変数のアドレス `A` に対する Read，Write の順序でのアクセスを含むトランザクション *Tx.X* を *thr.1* が，そして `A` に対する Read，Write の順序でのアクセスの後に更に `A` に対する Read アクセスを含むトランザクション *Tx.Z* を *thr.2* が，それぞれ実行している．まず，この例において *thr.2* が `load A` を実行した後に，*thr.1* が `load A` の実行を試みることで `A` に対するアクセスリクエスト *Req.A* を送信したとする（時刻 t1）．この時，*thr.2* は提案手法の動作に従い，*thr.1* に対して *Wait* リクエストを送信することで *thr.1* のアクセスを待機させる（t2）．その後，*thr.2* が

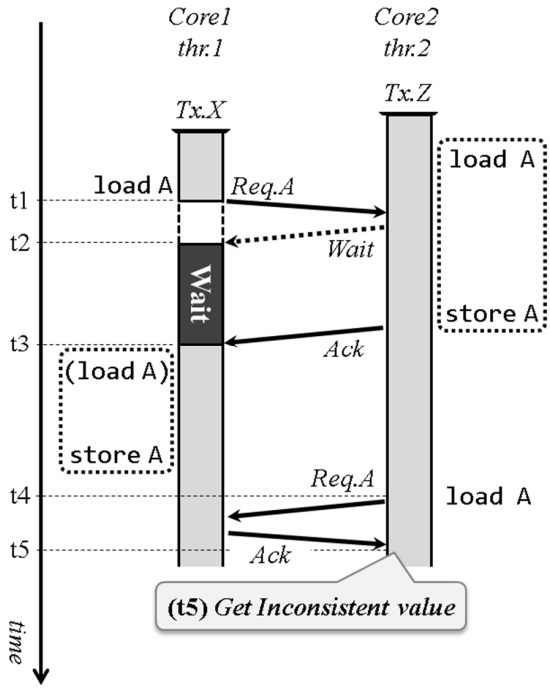


図 16: メモリ一貫性の欠如した状態に陥ってしまう例

`store A` を実行したとすると，*thr.2* は *thr.1* に *Ack* を送信することで *thr.1* の投機的アクセスを許可する（t3）．そして，*thr.2* によって投機的なアクセスを許可された *thr.1* が `store A` を実行した後に，*thr.2* が再度 `load A` の実行を試みたとする（t4）．この時，本来であれば *thr.2* は自身が *Tx.Z* 内で Write アクセスした A の値を用いて処理を進めなければならない．しかしこの例では，*thr.1* により Write アクセス済みとなった A の値を用いて，*thr.2* が *Tx.Z* の処理を進めることになってしまう．そのため，*thr.2* の実行する *Tx.Z* は 2.1 節で示したトランザクションの定義を満たさないことになり，*thr.1* と *thr.2* の間でメモリ一貫性が欠如した状態に陥ってしまう．

そこで，そのような場合は例外的に，Read，Write の順序でアクセスされる共有変数に対する Write アクセスの実行を完了したスレッドが他のスレッドによる投機的アクセスを許可せず，自身の実行トランザクションをコミットした後にこれを許可する．具体的な動作例を**図 17** に示す．この例では，図 16 の例と同様のトランザクション *Tx.X* および *Tx.Z* を，*thr.1* および *thr.2* が並列実行している．この例において，まず *thr.2* が `load A` を実行した後に，*Wait* リクエストにより *thr.1* のアクセスを待機させたとする（時刻 t1）．そして，*thr.2* が `store A` を実行することで，*thr.1* のアクセスが投機的に許可される（t2）．続いて，*thr.2* が `load A` の実行を試みたとすると，*thr.1* に

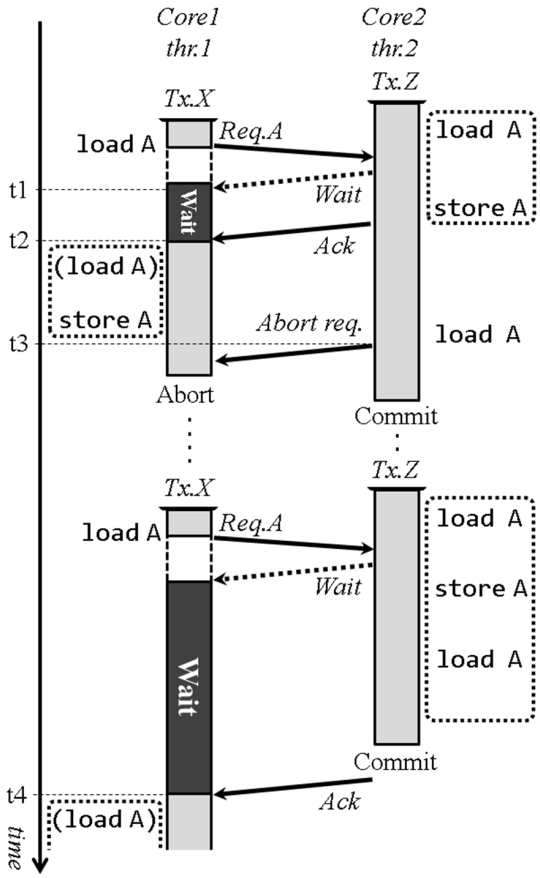


図 17: メモリ一貫性の欠如する問題に対する例外処理

より投機的にアクセスされた A に *thr.2* が再度アクセスすることになる．この時，*thr.2* はメモリ一貫性が欠如した状態に陥ってしまうことを防ぐために，*thr.1* に対して *Tx.X* のアボートを要求する（t3）．これ以降，*thr.2* は再び *Tx.Z* を実行した際に，A に対する Read，Write の順序でのアクセスが完了した時点では *thr.1* の投機的アクセスを許可せず，*Tx.Z* をコミットした後にこれを許可する（t4）．提案手法では，以上のように動作することでメモリ一貫性が欠如してしまうことを防いでいる．

次に，2 つめの問題であるデッドロック状態に陥ってしまう例を**図 18** に示す．この例では，共有変数のアドレス A に対する Read アクセスと，別の共有変数のアドレス B に対する Read アクセスを含むトランザクション *Tx.V* を *thr.1* が，そして A に対する Read，Write の順序でのアクセスと B に対する Write アクセスを含むトランザクション *Tx.W* を *thr.2* が，それぞれ実行している．まず，この例において *thr.1* が `load B` を，そして *thr.2* が `load A` をそれぞれ実行した後に，*thr.1* が `load A` の実行を試みたと

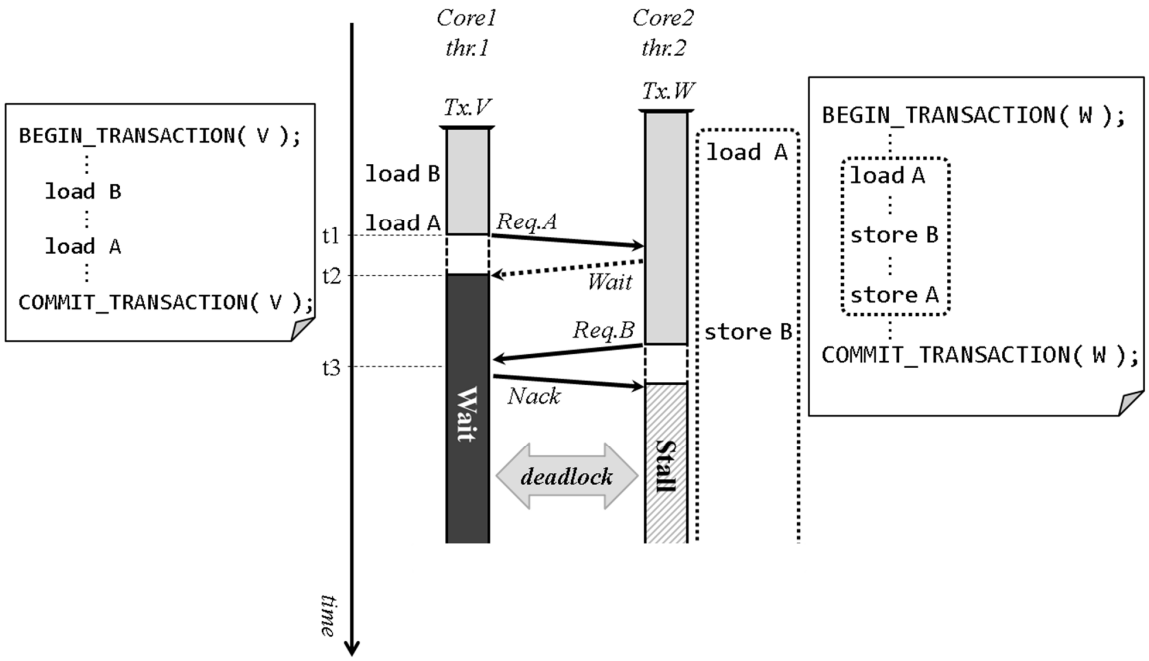


図 18: デッドロック状態に陥ってしまう例と各トランザクションの擬似コード

する（時刻 t1）．この時，*thr.2* は提案手法の動作に従い，*thr.1* に対して *Wait* リクエストを送信することで *thr.1* のアクセスを待機させる（t2）．その後，*thr.2* が `store` B の実行を試みたとすると，待機状態となっている *thr.1* は B において競合を検出し，*thr.2* に対して *Nack* を返信する（t3）．この *Nack* の受信により，*thr.2* は *Tx.W* をストールさせてしまうため，A に対する Write アクセスを実行する地点まで到達することができなくなる．したがって，*thr.1* のアクセスが *thr.2* によって許可されない状態で，*thr.2* が *Tx.W* をストールし続けるというデッドロック状態に陥ってしまう．

そこで，そのような場合は例外的に，待機スレッドの実行トランザクションをアボートさせることで，Read，Write の順序でアクセスされる共有変数に対する Read アクセスを実行したスレッドの処理を優先させるようにする．具体的な動作例を**図 19** に示す．この例では，*thr.1* および *thr.2* が図 18 の例と同様にトランザクション *Tx.V* および *Tx.W* を実行している．まず，この例において *thr.1* および *thr.2* が図 18 の例と同様の動作を行い，*thr.1* が競合を検出して *thr.2* に *Nack* を返信したとする（時刻 t3）．この時，*Nack* を受信した *thr.2* が A に対する Read アクセスを既に実行済みである場合，*thr.2* は *thr.1* に対してアボート要求を送信することで，*Tx.W* の処理を優先できるようにする．提案手法では，以上のように動作することでデッドロック状態が発生してしまうことを防いでいる．

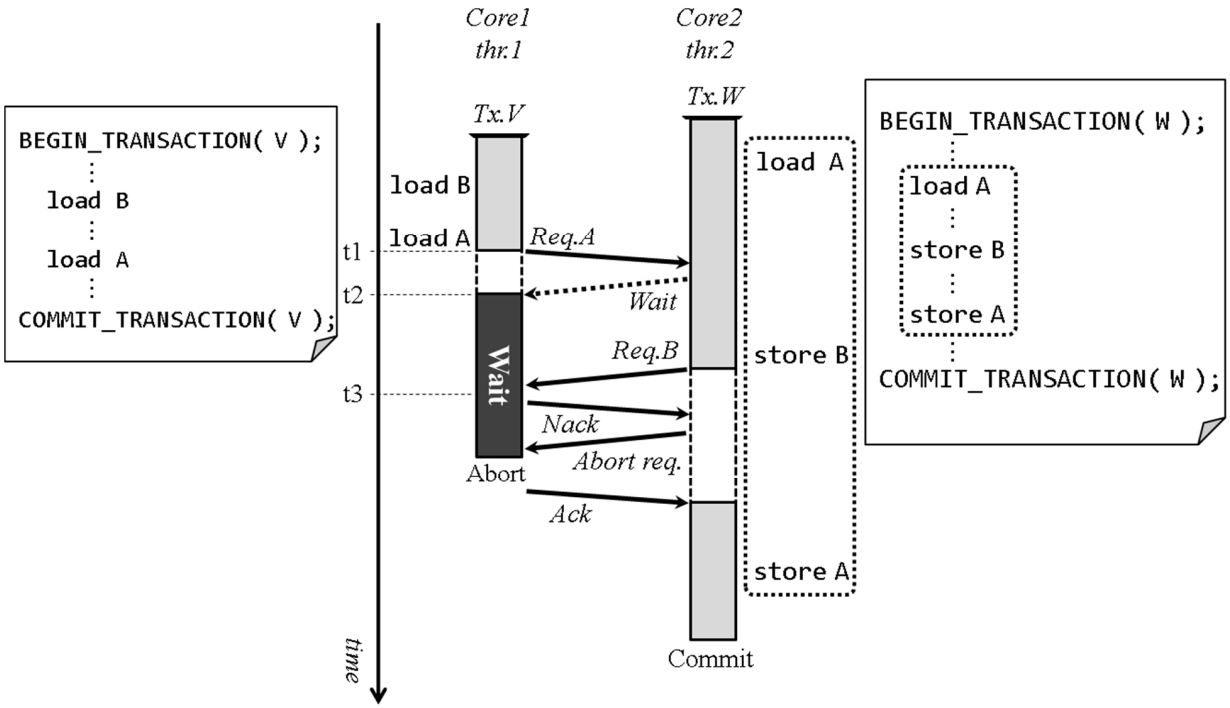


図 19: デッドロック状態が発生する問題に対する例外処理と各トランザクションの擬似コード

6 評価

これまでに述べた 2 つの提案手法を，HTM の研究で広く用いられている LogTM[22] に実装し，シミュレーションによる評価を行った．本章では，それぞれの提案手法の評価結果を示し，考察する．また，各提案手法を実現するために必要となるハードウェアコストおよびアクセスオーバヘッドを見積もる．

6.1 評価環境

評価には HTM の研究で広く用いられている Simics 3.0.31[23] と GEMS 2.1.1[9] の組み合わせを用いた．Simics は機能シミュレーションを行うフルシステムシミュレータであり，GEMS はメモリシステムの詳細なタイミングシミュレーションを担う．プロセッサ構成は 32 コアの SPARC V9 とし，OS は Solaris 10 とした．**表 2** に詳細なシミュレータ構成を示す．評価対象のプログラムとして GEMS microbench，SPLASH-2[10] および STAMP[11] から計 12 個を使用し，それぞれ 16 スレッドで実行した．ここで，各プログラムの実行時に用いた入力パラメタを**表 3** に示す．なお，STAMP に含まれ

表 2: シミュレータ諸元

Processor	SPARC V9
#cores	32 cores
clock	1 GHz
issue width	single
issue order	in-order
non-memory IPC	1
D1 cache	32 KBytes
ways	4 ways
latency	1 cycle
D2 cache	8 MBytes
ways	8 ways
latency	20 cycles
Memory	8 GBytes
latency	450 cycles
Interconnect network latency	14 cycles

る Kmeans と Vacation については，デフォルトで設定されている 2 種類の入力パラメタを用いて評価した.

6.2 評価結果

評価結果を**図 20**，**図 21** および**表 4** に示す．図 20 および図 21 では，各ベンチマークプログラムの評価結果がそれぞれ 4 本のバーで構成されている．これらのバーはそれぞれ左から順に，

(B) 既存の LogTM（ベースライン）

(R) 既存の LogTM で Store Predictor を用いた参考モデル

(P_1) 同一の共有変数に対する Read，Write の順序でのアクセスを含む処理を排他実行する提案モデル

(P_2) 同一の共有変数に対する Read，Write の順序でのアクセスが完了した時点で，投機的アクセスを許可する提案モデル

表 3: ベンチマークプログラムの入力パラメタ

GEMS microbench	
Btree	priv-alloc-20pct
Contention	config 1
Deque	4096ops 128bkoff
Prioqueue	8192ops
SPLASH-2	
Barnes	512bodies
Cholesky	tk14.0
Radiosity	-p 31
Raytrace	teapot
STAMP	
Kmeans	random-n2048-d16-c16.txt
Kmeans+	random-n16384-d24-c16.txt
Vacation	-n4 -q60 -u90 -r16384 -t4096
Vacation+	-n4 -q60 -u90 -r1048576 -t4096

の実行サイクル数の平均を表しており，既存モデル (B) の実行サイクル数を 1 として正規化している．なお，フルシステムシミュレータ上でマルチスレッドによる動作シミュレーションを行う際には，性能のばらつきを考慮する必要がある [24]．したがって，各対象につき試行を 10 回繰り返し，得られた結果から 95%の信頼区間を求めた．信頼区間は図中にエラーバーで示す．なお，3 章でも述べたように，参考モデル (R) で用いる Store Predictor には Read，Write の順序でアクセスされるアドレスが記憶される．本評価では，この参考モデル (R) の理想的な性能を評価するために，ベンチマークプログラム中で出現する，Read，Write の順序でアクセスされる全てのアドレスを Store Predictor に記憶できる状況における，参考モデル (R) の性能を評価した．なお，図 20 では全モデルと全プログラムの組み合わせの評価結果を示しているが，参考モデル (R) で Btree を実行した場合のサイクル数が非常に大きい値となったため，共通のスケールでは他のサイクル数およびその内訳を判別しづらくなっている．そこで，図 20 から，参考モデル (R) で Btree を実行した結果のみを除いたものを図 21 として別に示す．ここで，図 20 および図 21 の凡例は以下のように実行サイクル数の内訳を示し

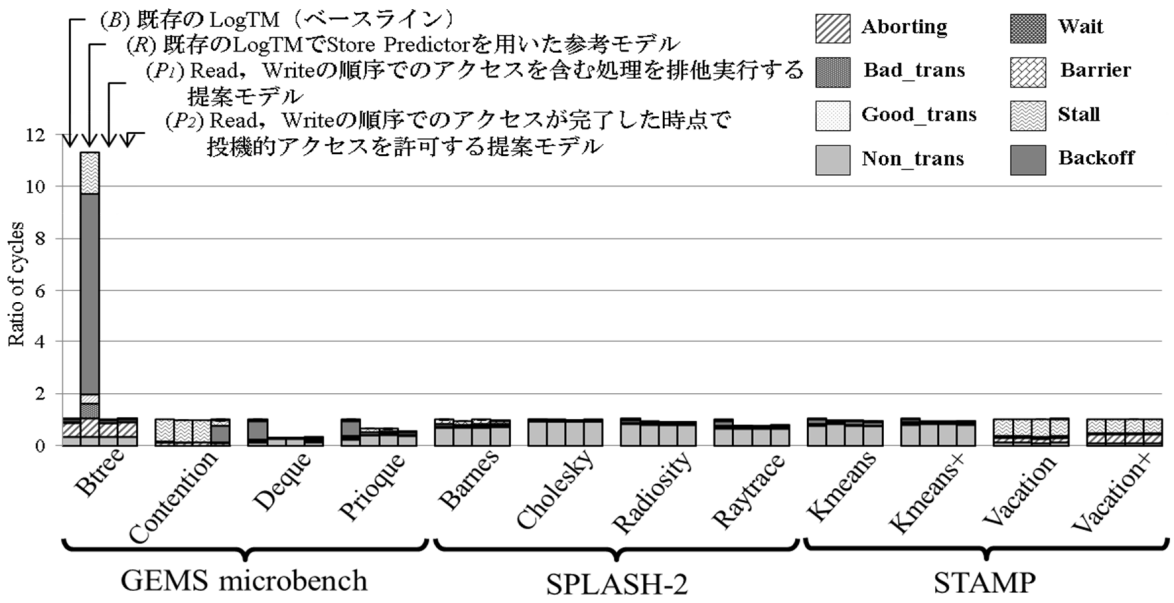


図 20: 各プログラムの実行サイクル数比

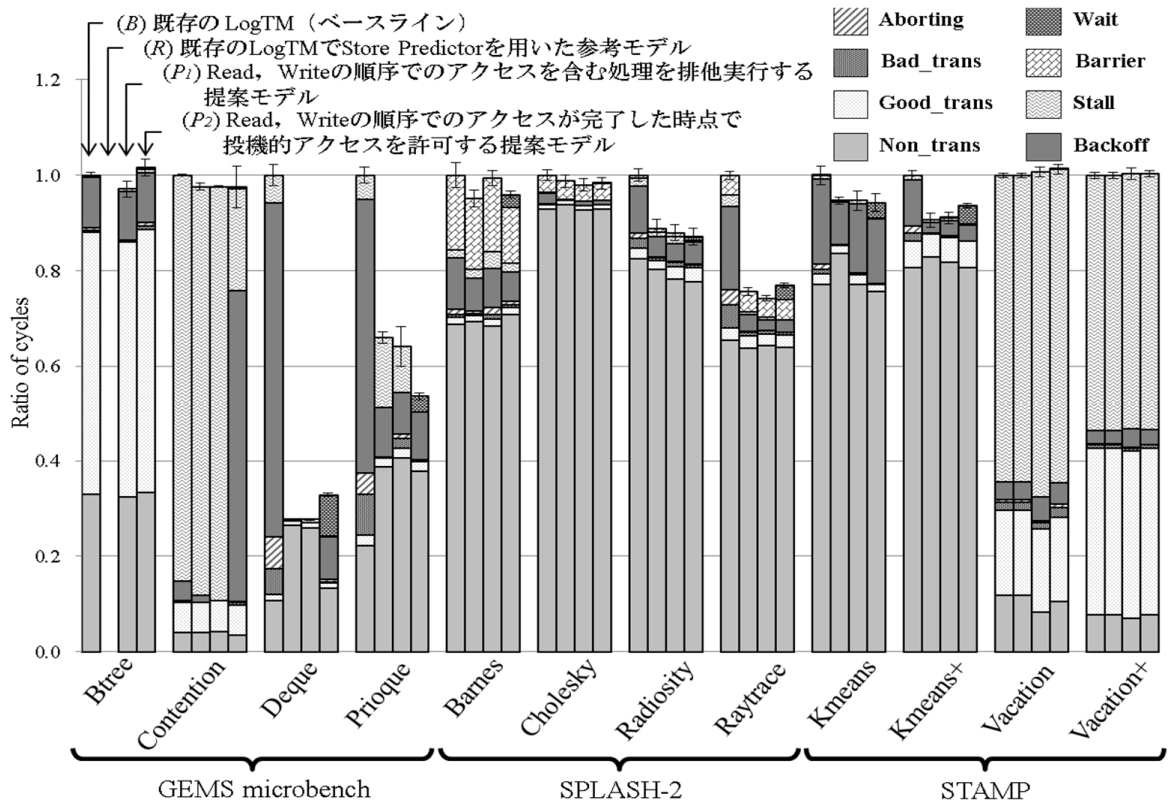


図 21: 各プログラムの実行サイクル数比（参考モデル (R) による Btree の評価を除く）

表 4: 各ベンチマークプログラムにおけるサイクル削減率

		GEMS microbench	SPLASH-2	STAMP	All
(P_1)	平均	28.4%	9.4%	3.0%	13.6%
	最大	72.3%	25.7%	7.5%	72.3%
(P_2)	平均	28.6%	10.4%	2.7%	13.9%
	最大	67.2%	23.1%	6.4%	67.2%

ている．

Wait: 提案手法で追加した待機処理に要したサイクル数

Barrier: バリア同期に要したサイクル数

Stall: ストールに要したサイクル数

Backoff: バックオフ処理に要したサイクル数

Aborting: アボート処理に要したサイクル数

Bad_trans: アボートされたトランザクションの実行サイクル数

Good_trans: コミットされたトランザクションの実行サイクル数

Non_trans: トランザクション外の実行サイクル数

なお，アボート直後にトランザクションが再開された場合，競合が再発してしまう可能性がある．そのような競合の再発を抑制する手段として，既存の LogTM では Exponential Backoff というアルゴリズムが採用されている．このアルゴリズムにより算出される待機時間は，アボートが繰り返されるごとに指数関数的に増大する．凡例の Backoff は，この待機時間の総和を表している．

まず，図 20 に示す評価結果のグラフを見てみると，Btree において参考モデル (R) の性能が大幅に低下していることが見てとれる．その一方で，参考モデル (R) の Btree における評価を除いた図 21 のグラフを見てみると，2 つの提案モデル (P_1)，(P_2) の性能が既存モデル (B) と比較して向上，またはほぼ変わっていないことが分かる．このことから，参考モデル (R) で性能が大きく悪化してしまうようなプログラムに対しても，2 つの提案モデル (P_1) および (P_2) は有効であることが確認できた．なお，このように Btree において大きな性能差が生じる理由については，後の 6.3 節で詳細に述べる．

また，図 21 のグラフでは，ほぼ全てのプログラムで 2 つの提案モデル (P_1) および (P_2) は既存モデル (B) と比較して，大幅に性能向上していることが分かる．このことから，これらの提案モデルでは Futile Stall とそれに起因するアボートを十分に抑制で

```
1 BEGIN_TRANSACTION( 16 );
2   ray->id = gm->rid++;
3 COMMIT_TRANSACTION( 16 );
```

図 22: Raytrace プログラム内のトランザクション

きることが確認できた．ここで，2 つの提案モデルの性能向上率をまとめると，1 つめの提案モデル (P_1) は既存モデル (B) に対して最大 72.3%，平均 13.6%の性能向上を達成できており，2 つめの提案モデル (P_2) は既存モデル (B) に対して最大 67.2%，平均 13.9%の性能向上を達成できた．

6.3 考察

本節では，2 つの提案手法の評価結果を用いて詳細な考察を行う．

6.3.1 同一変数に対する Read，Write の順序でのアクセスを排他実行するモデル

まず，1 つめの提案モデル (P_1) と既存モデル (B) を比較すると，1 つめの提案モデル (P_1) は多くのプログラムで性能向上を達成できていることが分かる．特に，Deque，Prioque，Radiosity，Raytrace，Kmeans および Kmeans+において，Backoff サイクルが大幅に削減されている．これらのプログラム中には，同一の共有変数に対する Read，Write の順序でのアクセスを行うトランザクションが含まれている．そのため，既存モデル (B) ではそのようなトランザクションが複数のスレッドによって並列実行されることで Futile Stall やそれに起因するアボートが繰り返し発生し，Exponential Backoff による Backoff サイクルの増大が引き起こされていた．提案モデル (P_1) では，このような Futile Stall やそれに起因するアボートの繰り返しを十分抑制することができたために，Backoff サイクルが大幅に削減されたと考えられる．ここで Raytrace のプログラム中に存在する，同一の共有変数に対する Read，Write の順序でのアクセスを含むトランザクションの 1 つを**図 22** に示す．このトランザクション内では，共有変数 `gm->rid` に対するインクリメントが行われており，これが同一変数に対する Read，Write の順序でのアクセスに相当する．このインクリメントはユニークな ID を生成するために排他的に実行されなければならないが，複数スレッドがこの操作を並列に実行した場合，既存モデル (B) において Futile Stall とそれに起因するアボートが繰り返し発生し，Backoff サイクルが増大してしまう．一方，提案モデル (P_1) では図 22 に示したトランザクションに含まれる Read，Write の順序でのアクセスを排他的に実行し，上記の問

```
1  BEGIN_TRANSACTION( 0 );
2  Btree_insert (*node, insert, key, *ptr) {
3    if (node->isLeaf) {
4      node->key_num++;
5      node->child[node->key_num] = node->child[node->key_num-1];
6      for (i = node->key_num; i >= insert+1; --i) {
7        if (i != NODE_NUM_PAIRS)
8          node->key[i] = node->key[i - 1];
9        node->child[i] = node_child[i - 1];
10     }
11     if (insert < NODE_NUM_PAIRS)
12       node->key[insert] = key;
13     node_child[insert] = ptr;
14   }
15 }
16 COMMIT_TRANSACTION( 0 );
```

図 23: Btree プログラム内の挿入トランザクション（$Tx.I$）

題を抑制したことが Backoff サイクルの減少に寄与したと考えられる．次に，Cholesky を見てみると，提案モデル (P_1) においてあまり大きな性能向上を達成できていないことが分かる．この理由として，Cholesky ではトランザクション外のサイクル数である Non_trans が総実行サイクル数の大半を占めているため，提案モデル (P_1) による性能向上の比率が小さくなったことが挙げられる．一方，Vacation および Vacation+では提案モデル (P_1) において性能向上が確認できないが，これは，これらのプログラムに同一の共有変数に対する Read，Write の順序でのアクセスが含まれていないからである．

ここで，1 つめの提案モデル (P_1) と Store Predictor を用いた参考モデル (R) を比較する．まず Btree では，6.2 節で述べたように，参考モデル (R) の性能が大幅に低下していることが図 20 から見てとれる一方で，図 21 からは 1 つめの提案モデル (P_1) の性能が向上していることが分かる．この Btree というプログラムには，**図 23** および **図 24** に示すような，ツリー構造に対する挿入および検索のための 2 種類のトランザクション（仮に $Tx.I$，$Tx.L$ とする）が存在している．このうち，ツリー構造に対する挿入処理である $Tx.I$ には共有変数 node->key_num に対する Read，Write の順序で

```
1  BEGIN_TRANSACTION( 1 );
2  Btree_lookup (*node, key) {
3    if (node->isLeaf) {
4      for (i = 0; i < node->key_num; ++i) {
5        if (key <= node->key[i])
6          index = i;
7      }
8    } else {
9      for (i = 0; i < node->key_num; ++i) {
10       if (key > node->key[i])
11         index = i;
12     }
13   }
14   node = node->child[index];
15 }
16 COMMIT_TRANSACTION( 1 );
```

図 24: Btree プログラム内の検索トランザクション（$Tx.L$）

のアクセスが含まれている（4 行目）が，検索処理である $Tx.L$ には当該共有変数に対する Read アクセスしか含まれていない．そのため，$Tx.I$ が並列実行される際には 1 つめの提案モデル (P_1) は有効であり，実際に内訳を比較してもこのモデルは既存モデル (B) に対して Backoff サイクルが減少している．しかし，$Tx.L$ が並列実行される場合には，4.2.4 項で述べたような Read アクセスしか行わないトランザクションの逐次実行による性能低下が引き起こされる．参考モデル (R) では，この $Tx.L$ で共有変数 node->key_num に対して Read アクセス済みとなったスレッドが存在する場合，他のスレッドが $Tx.L$ で当該共有変数に Read アクセスを試みたとしても，$Nack$ が返信されてしまう．そのため，本来であれば $Tx.L$ を並列実行できる場面で，1 つのスレッドが複数スレッドの実行トランザクションをストールさせてしまうため，Stall サイクルが大幅に増大している．また，実行トランザクションをストールさせるスレッドが多くなるほど，デッドロックの発生する可能性が高くなり，それにともなってそのデッドロックを解消するアボートが頻繁に発生してしまうため，Backoff サイクルが大幅に増大している．これに対して，1 つめの提案モデル (P_1) では，Read アクセスしか行わ

```
1  BEGIN_TRANSACTION( 18 );
2    if (gm.lookup_array[key] != NOPE) {
3      /* キュー内の各要素に対する操作 */
4      gm.array[index].element += 1;
5      /* キュー内の各要素に対する操作 */
6      if (gm.lookup_array[tmp_l] != index) {
7        gm.lookup_array[tmp_l] = index;
8      }
9    }
10 COMMIT_TRANSACTION( 18 );
```

図 25: Prioque プログラム内のトランザクション

れなかった際に C ビットをクリアするようにしたことで，このような性能低下を防ぐことができた．この Btree のように，あるデータ構造に対する挿入/検索処理を含むトランザクションが並列に実行される際には，逐次実行による性能低下が引き起こされる可能性が高く，データ構造が複雑でトランザクション処理時間がさらに長くなった場合に，特にその性能低下が顕著となってしまう．このことから，HTM においてトランザクションのスケジューリングを行う際には，1 つめの提案モデル (P_1) のように Read アクセスしか行わないトランザクションを考慮することが重要である．

次に，図 21 で Prioque に着目すると，どちらのモデルも既存モデル (B) に対して性能向上している一方で，参考モデル (R) の Stall サイクルが 1 つめの提案モデル (P_1) よりも増大している．この理由として，Prioque には条件分岐により必ずしも Read，Write の順序でアクセスされない共有変数が存在することが挙げられる．ここで，Prioque に存在するトランザクションの 1 つを**図 25** に示す．図中のトランザクションでは，最初の if 文で配列 `gm.lookup_array[]` に対する Read アクセスが行われた後に（2 行目），キュー内の各要素に対する操作の中で共有変数 `gm.array[].elemnt` に対する Read，Write の順序でのアクセスが行われる（5 行目）．1 つめの提案モデル (P_1) と参考モデル (R) では，この Read，Write の順序でのアクセス間の処理を並列実行することにより発生するアボートを抑制することで既存モデル (B) に対する性能向上を達成できた．ここで，図中のトランザクションでは，8 行目の if 文による条件分岐においてその条件式が真である場合にのみ，配列 `gm.lookup_array[]` に対する Write アクセスが行われる（8，9 行目）．そのため，配列 `gm.lookup_array[]` は必ずしも Read，

Write の順序でアクセスされず，このトランザクションを複数スレッドが並列実行したとしても，配列 `gm.lookup_array[]` に対するアクセスに起因するアボートが発生する可能性は低いと考えられる．しかし，参考モデル (R) では一度でも Read，Write の順序でアクセスされるアドレスは，Store Predictor に記憶されてしまう．そのため，参考モデル (R) において配列 `gm.lookup_array[]` のアドレスが Store Predictor に記憶されることで，2 行目の if 文からの処理が全て逐次実行されていた．これにより，このトランザクションを実行するスレッドと競合したスレッドは，より長い間トランザクションをストールすることになるため，Stall サイクルが増大してしまったと考えられる．ただし，1 つめの提案モデル (P_1) であっても，このような条件分岐により処理内容が変化してしまう場合を考慮した動作は実現できていないため，今後はこのような条件分岐を考慮したスケジューリング手法を検討する必要がある．

また，Barnes では参考モデル (R) の性能が向上している一方で，1 つめの提案モデル (P_1) の性能が向上していない．これは，同一の共有変数に対する Read，Write の順序でのアクセスを含むトランザクションの直後に，当該変数に対して Read アクセスしか行わない可能性のあるトランザクションが存在することが原因であると考えられる．このような挙動を引き起こしうるトランザクションを**図 26** に示す．図中では，まず最初のトランザクション内で共有変数`*qptr` に対する Read，Write の順序でのアクセスが実行される（2 行目）ため，1 つめの提案モデル (P_1) ではこの時点で当該変数に対応する C ビットがセットされる．続いて，最初のトランザクションの直後に存在するトランザクション内で共有変数`*qptr` のノードタイプが `LEAF` であるかどうかで条件分岐が発生する（8 行目）．この時，この分岐の条件式が偽であった場合，共有変数`*qptr` に対する Read アクセスのみが実行された後にトランザクションが即座にコミットされるため，2 行目の時点でセットされた C ビットがここでクリアされてしまう．このように，Barnes では共有変数`*qptr` に対応する C ビットがセットされたとしても，比較的短い期間のうちにこの C ビットがクリアされてしまう状況が多く発生していた．これにより，1 つめの提案モデル (P_1) では同一変数に対する Read，Write の順序でのアクセスが排他実行されなかったため，性能が向上しなかったと考えられる．

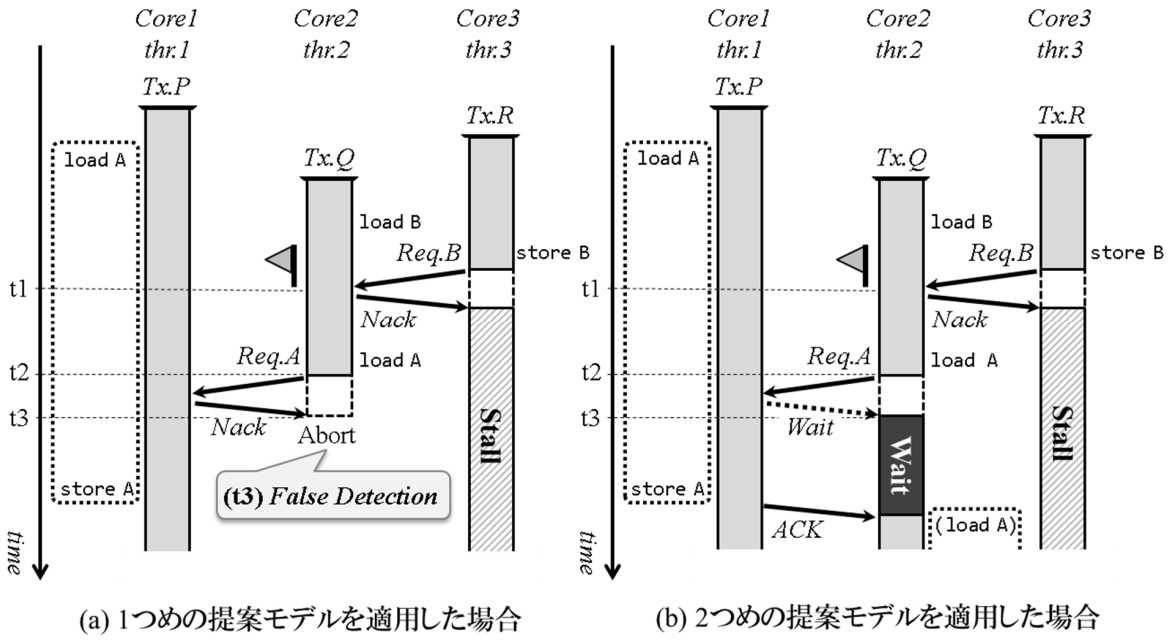
6.3.2 同一変数に対する Read，Write の順序でのアクセスを完了した時点で投機的アクセスを許可するモデル

次に，1 つめの提案モデル (P_1) と 2 つめの提案モデル (P_2) を比較すると，Prioque および Radiosity において 2 つめの提案モデル (P_2) の方が 1 つめの提案モデル (P_1) よりも性能向上している．この理由として，これらのプログラムでは 1 つめの提案モデ

```
1  BEGIN_TRANSACTION( 2 );
2  /* 共有変数 *qptr に対する Read, Write の順序でのアクセス */
3  COMMIT_TRANSACTION( 2 );
4
5  ...
6
7  BEGIN_TRANSACTION( 3 );
8    if (Type(*qptr) == LEAF) {
9      le = (leafptr) *qptr;
10     if (le->num_bodies == MAX_BODIES_PER_LEAF) {
11       *qptr = (nodeptr) SubdivideLeaf(le, (cellptr) mynode, l,
       ProcessId);
12     } else {
13       Parent(p) = (nodeptr) le;
14       Level(p) = l;
15       ChildNum(p) = le->num_bodies;
16       Bodyp(le)[le->num_bodies++] = p;
17       flag = FALSE;
18     }
19   }
20 COMMIT_TRANSACTION( 3 );
```

図 26: Barnes プログラム内のトランザクション

ル (P_1) においてデッドロックの誤検出によるアボートが頻繁に発生していたことが挙げられる．ここで，そのような問題が発生してしまう例を，**図 27**(a) に示す．この例では，トランザクション *Tx.P*，*Tx.Q* および *Tx.R* を，3 つのスレッド（*thr.1*〜*thr.3*）がそれぞれ実行している．はじめに，*thr.2* が自身よりもトランザクション開始時刻の早い *thr.3* に対して *Nack* を返信した際に，2.3 節で述べたように possible_cycle フラグがセットされる（時刻 t1）．その後，*thr.2* は `load A` の実行を試みるが（t2），この時点では既に *thr.1* が `load A` を実行しているため，*thr.1* は 1 つめの提案モデル (P_1) の動作に従って *thr.2* に *Nack* を返信する．この時，*thr.2* は possible_cycle フラグをセットした状態で，自身よりも早くトランザクションを開始した *thr.1* から *Nack* を受信す



(a) 1つめの提案モデルを適用した場合　　(b) 2つめの提案モデルを適用した場合

図 27: デッドロックの誤検出によるアボート

ることでデッドロックが発生したと誤検出し，*Tx.Q* をアボートしてしまう（t3）．これに対し，同様の状況で 2 つめの提案モデル (P_2) の動作を適用した例を図 27(b) に示す．この例では，時刻 t3 において *thr.2* が *thr.1* から *Nack* ではなく *Wait* リクエストを受信することでデッドロックの誤検出による *thr.2* のアボートを防ぐことができ，さらに *thr.1* が *thr.2* の投機的アクセスを許可できる．Prioque と Radiosity では，上述したような動作が頻繁に発生したため，2 つめの提案モデル (P_2) が 1 つめの提案モデル (P_1) よりも高い性能を達成できたと考えられる．

続いて，Btree，Contention，Deque，Raytrace および Kmeans+の 4 つのプログラムでは，1 つめの提案モデル (P_1) の方が 2 つめの提案モデル (P_2) よりも性能向上している．この理由として，これらのプログラムでは 5.3 節で示した 2 つめの提案モデル (P_2) の例外処理である，デッドロック状態を回避するためのアボートが頻繁に発生していたことが挙げられる．ここで，そのような問題が発生してしまう例を**図 28** に示す．この例では，トランザクション *Tx.J* および *Tx.K* を，2 つのスレッド *thr.1* および *thr.2* がそれぞれ実行しており，*thr.1* が *thr.2* によって `load A` の実行を投機的に許可された後に（時刻 t1），*thr.1* が *thr.2* よりも早くトランザクションのコミットに到達することで *thr.2* のコミットを待ち続けている（t2）．このような状況で，*thr.2* が `store B` の実行を試みたとすると，この時点では既に *thr.1* が `load B` を実行済みで

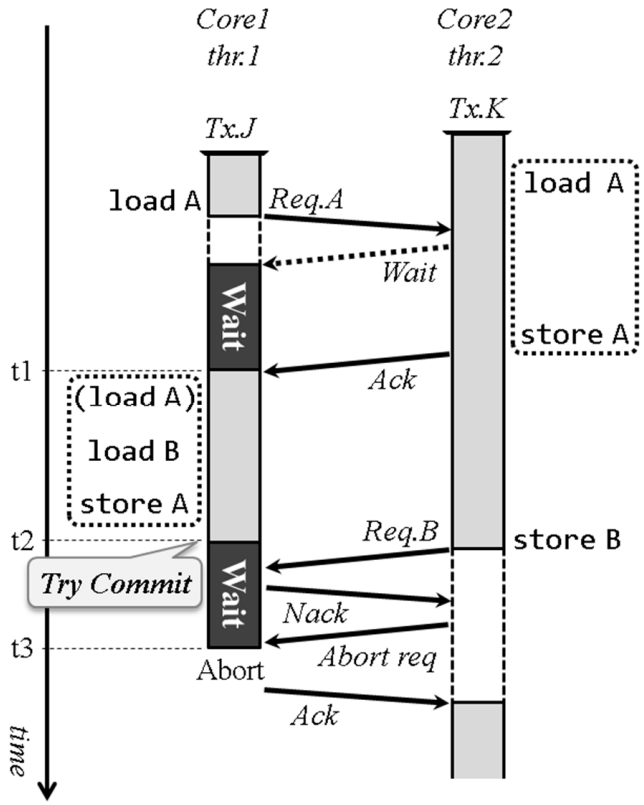


図 28: デッドロック状態を回避するための例外処理に起因するアボート

あることから，*thr.1* は *thr.2* に対して *Nack* を返信する．この *Nack* により，これらのスレッド間でデッドロック状態が発生するため，*thr.2* は例外処理に従って *thr.1* に対してアボートリクエストを送信し，*thr.1* の *Tx.J* を結果としてアボートさせてしまう（t3）．2 つめの提案モデル (P_2) では，上記のような動作に起因するアボートの繰り返しによって，1 つめの提案モデル (P_1) よりも性能が低下してしまったと考えられる．したがって，今後はプログラムごとにアクセスパターンを詳細に調査し，投機的アクセスの許可対象とする変数をより適切に決定できる枠組みを検討する必要がある．

6.4　ハードウェアコストとアクセスオーバヘッド

本節では，2 つの提案手法を実現するために追加したハードウェアの実装コストとそのアクセスオーバヘッドについて述べる．

6.4.1　ハードウェアコスト

まず，1 つめの提案手法で追加した C ビットは当該キャッシュラインが Read，Write の順序でアクセスされたか否かを示すため，1 ラインあたり 1bit だけ必要となる．な

お，本論文の評価環境ではL1キャッシュサイズが32KBytes，キャッシュラインサイズが64Bytes，そしてウェイ数が4であることから，L1キャッシュの行数は128行となる．したがって，この提案手法を実装するために追加されるハードウェアコストは16コア構成のプロセッサにおいて，1bit × 128lines × 16cores = 256Bytes とごく少量である．

次に，2つめの提案手法では上述したCビットに加え，Lビットと Address Table および Dependence Table の2つの表が追加される．このうち，Address Table と Dependence Table には，各トランザクションで Futile Stall とそれに起因するアボートが引き起こされたアドレスを全て記憶できるだけのエントリ数が必要となる．さらに，Dependence Table には，ある共有変数に対する複合操作が含まれているトランザクションの最大数だけ，Target-TxID と Target-PC を記憶するフィールドが必要となる．そこで，各プログラムに2つめの提案手法の動作を適用した際に，上述したエントリ数とトランザクション数がどの程度になるのかを調査した．この結果を**表5**および**表6**に示す．これより，Address Table は10行のエントリ，そして Dependence Table は10行のエントリおよび3組の Target-TxID と Target-PC を記憶するフィールドがあれば，今回評価に用いたプログラムでは情報を過不足なく記憶できることが分かる．ここで，Address Table の1つのエントリで必要となる記憶容量は，1つの Target-Address に対して 64bits である．一方，Dependence Table の1つのエントリで必要となる記憶容量は Prev-Core，Next-Core，Target-TxID，Target-PC に対してそれぞれ 4bits，4bits，4bits，64bits である．なお，Address Table は格納されているアドレスを高速に検索する必要があるため CAM で構成する．これに対し，Dependence Table の各エントリは Address Table の各エントリと一対一に対応しており，Address Table のインデクスを用いて高速に検索が行うことができるため RAM で構成する．つまり，Address Table は1行あたり 64bits の幅を持つエントリが10行ある CAM で構成でき，Dependence Table は1行あたり 4bits + 4bits + (4bits + 64bits) × 3 = 212bits の幅を持つエントリが10行ある RAM で構成できる．また，この提案手法ではL1キャッシュラインに対してCビットとLビットを追加するため，1ラインあたり 2bits のフィールドが1つめの提案手法と同様に128行分だけ必要となる．したがって，2つめの提案手法を実装するために必要となるハードウェアコストは，16スレッドを実行可能な16コア構成のプロセッサにおいて約 6KBytes となり，1コアあたり約 350Bytes となる．この 350Bytes という数値は1コアあたりの L1 キャッシュサイズである 32KBytes と比較しても十分に小さいものである．

表 5: Futile Stall に起因するアボートが引き起こされたアドレス数

GEMS microbench			
Btree	Contention	Deque	Prioque
9	2	4	6
SPLASH-2			
Barnes	Cholesky	Radiosity	Raytrace
10	6	10	10
STAMP			
Kmeans	Kmeans+	Vacation	Vacation+
7	9	3	3

表 6: 同一変数への Read，Write の順序でのアクセスを含むトランザクションの最大数

GEMS microbench			
Btree	Contention	Deque	Prioque
2	1	1	1
SPLASH-2			
Barnes	Cholesky	Radiosity	Raytrace
2	2	3	2
STAMP			
Kmeans	Kmeans+	Vacation	Vacation+
1	1	2	2

6.4.2 アクセスオーバヘッド

本項では，2 つめの提案モデル (P_2) で追加した 2 つの表のアクセスオーバヘッドが性能に及ぼす影響について述べる．まず，2 つの表のアクセスオーバヘッドはそれぞれの表を参照した総回数 C，そして参照時のレイテンシ T を用いて，$C \times T$ として概算する．ここで，各プログラムを 2 つめの提案モデル (P_2) で実行した際に，それぞれの表が参照された総回数を**表 7** および**表 8** に示す．なお，6.4.1 項でも述べたとおり，Address Table は 10 行のエントリを持つ CAM で構成されるため，この表を一般的な TLB と同じ 1cycle のレイテンシで参照できると仮定する．一方，Dependence Table は 212bits の幅を持つエントリが 10 行ある RAM で構成される．この Dependence Table

表 7: Address Table の総参照回数

GEMS microbench			
Btree	Contention	Deque	Prioque
3,037	15,760	12,788	41,966
SPLASH-2			
Barnes	Cholesky	Radiosity	Raytrace
3,360	7,370	31,708	53,706
STAMP			
Kmeans	Kmeans+	Vacation	Vacation+
8,673	46,062	12,277	20,786

表 8: Dependence Table の総参照回数

GEMS microbench			
Btree	Contention	Deque	Prioque
5,750	30,791	21,126	78,345
SPLASH-2			
Barnes	Cholesky	Radiosity	Raytrace
6,303	14,403	61,851	100,416
STAMP			
Kmeans	Kmeans+	Vacation	Vacation+
16,830	90,195	24,544	41,537

の各エントリを参照する際には，Address Table のインデクスから対象のエントリを検索するのに 1cycle が，そして Dependence Table の各フィールドに対するマスク操作と比較操作にそれぞれ 1cycle が必要となると仮定する．また，Dependence Table には 3 組のトランザクション ID とプログラムカウンタの値が格納されていることから，各フィールドに対するマスク操作と比較操作が最大で 3 回行われることになる．したがって Dependence Table は，最大で $1\text{cycle} + (1\text{cycle} + 1\text{cycle}) \times 3 = 7\text{cycles}$ のレイテンシで参照できる．これらのことから，2 つの表のアクセスオーバヘッドが各ベンチマークプログラムの総実行サイクル数に占める割合を算出したところ，その割合が最大となる Prioque でも 0.89%となり，非常に小さなものであることが確認できた．な

お，このオーバヘッドは 2 つの表の構成や動作アルゴリズム次第でさらに小さなものにできると考えられるため，今後はこれらの表の構成等を工夫する必要がある．

7　おわりに

本論文では，同一の共有変数に対する Read，Write の順序でのアクセスに起因する問題である，Futile Stall を解決する 2 つの手法を提案した．まず，1 つめの提案手法では，実行トランザクション内で同一変数に対して Read，Write の順序でアクセスしたスレッドが，当該トランザクションをコミットするまで他スレッドによる当該変数へのアクセスを許可しないことで排他的な実行を実現した．この手法により，HTM の性能を低下させる Futile Stall およびそれに起因するアボートの抑制を目指した．また 2 つめの提案手法では，各スレッドが実行トランザクション内で同一変数に対する Read，Write の順序でのアクセスを完了した時点で，当該トランザクションのコミットに先立って他のスレッドによる当該変数への投機的なアクセスを許可した．この手法により，1 つめの提案手法のさらなる高速化を目指した．

提案した 2 つの手法の有効性を確認するために，既存の HTM を拡張し，GEMS microbench，SPLASH-2 および STAMP を用いてシミュレーションによる評価を行った．評価の結果，1 つめの提案手法は既存の HTM と比較して，16 スレッド実行時で最大 72.3%，平均 13.6%の高速化を達成できることを確認した．一方，2 つめの提案手法は既存の HTM と比較して，16 スレッド実行時で最大 67.2%，平均 13.9%の高速化を達成でき，1 つめの提案手法と比較しても高速化を達成できるプログラムがあることを確認した．また，2 つの提案手法を実現するために必要な追加ハードウェアのコストを概算したところ，そのコストは 1 つめの提案手法で 256Bytes，そして 2 つめの提案手法で約 6KBytes と，少量であることを確認した．

しかし，1 つめの提案手法では，同一変数に対する Read，Write の順序でのアクセスを含むトランザクションの直後に，その変数に対して Read アクセスしか行わない可能性のあるトランザクションが存在する場合，対応する C ビットがセットされたとしても比較的短い期間のうちにその C ビットがクリアされてしまうことがある．これにより，同一変数に対する Read，Write の順序でのアクセスを含む処理を排他実行する動作が適用されず，性能が向上しない場面が見られた．したがって今後は，C ビットをクリアするためのアルゴリズムを改良していく必要がある．また 2 つめの提案手法では，投機的アクセスの許可に起因するメモリ一貫性の欠如した状態の発生を防ぐための例外処理を設定したが，この例外処理では同一変数に対する Read，Write の順序

でのアクセスの後にさらに当該変数に対するアクセスを含むようなトランザクションが実行された場合，トランザクションがコミットされるまで他のスレッドが当該変数にアクセスできず，待機処理が増大してしまう可能性がある．しかし，仮に同一変数に対する Read，Write の順序でのアクセスの後に存在するアクセスが Read アクセスではなく Write アクセスであった場合，その Write アクセスが実行される地点を同一変数に対する Read，Write の順序でのアクセスが完了する地点として新たに記憶することで，無駄な待機処理を削減できる余地がある．したがって，今後はベンチマークプログラムごとにアクセスパターンを詳細に調査し，同一の共有変数に対する Read，Write の順序でのアクセスが完了する地点をより厳密に記憶できる枠組みを検討する必要がある．

謝辞

本研究を進めていくにあたり，時には厳しく，時には優しくご指導してくださった津邑公暁 准教授に深く感謝致します．津邑公暁 准教授の慈愛に満ちたお言葉があったからこそ，私は修士論文を完成させるステージまで到達できたのだと思っています．最後まで丁寧にご指導していただき，本当に本当にありがとうございました．このご恩は一生かけて返していきたいと思っています．また，私をいつも優しい眼差しで見守ってくださった松尾啓志 教授，齋藤彰一 准教授，松井俊浩 准教授，梶岡慎輔 助教，川島龍太 助教に深く感謝致します．そして，私のような者を研究室の仲間として歓迎してくださった松尾・津邑研究室，齋藤研究室，松井研究室の方々に深く感謝致します．特に，CAMP の M2 の方々にはどれだけ感謝してもしきれません．まず，その溢れでる才能と能力により，私に的確なアドバイスやツッコミをしてくさだった井手上慶氏に深く感謝致します．あなたが野良猫を可愛がる姿を見て，犬派だった私が猫派になったことはとても心に残っています．また，CAMP のエースの座を絶対のものとし，私に様々な刺激を与えてくださった柴田裕貴氏に深く感謝致します．常に物事のギリギリを攻めるあなたの姿に，私はただただ驚かされるばかりでした．そして，強烈なギャグセンスとハイテンションで，私のつまらないボケを何度も救ってくださった松永拓也氏に深く感謝致します．深夜に研究室で，あなたとともに曲作りに励んだことは一生忘れません．ここで，CAMP の M2 ではありませんが，特別に松尾研究室の福田諭氏に深く感謝致します．あなたと交わした髭に関する深い議論は，今後の私の人生に新たな方向性を示してくれました．最後となりますが，津邑研究室で過ごした三年間の想い出は，私にとって一生忘れないかけがえのない宝物です．本当に，本

当にありがとうございました.

著者発表論文

論文

1. Ryohei YAMADA, Koshiro HASHIMOTO, Tomoaki TSUMURA, : “Priority-Based Conflict Resolution for Hardware Transactional Memory”, Proc. 2nd Int’l Workshop on Computer System and Architectures (CSA’14), held in conlunction with CANDAR’14, REGULAR PAPER, Shizuoka, Japan, pp.433-439 (Dec. 2014)
2. 橋本 高志良, 堀場 匠一朗, 江藤 正通, 津邑 公暁, 松尾 啓志, : “Read-after-Read アクセスの制御によるハードウェアトランザクショナルメモリの高速化”, 情報処理学会論文誌 コンピューティングシステム, Vol.6, No.4 (ACS44), pp.58-71 (Oct. 2013)
3. Koshiro HASHIMOTO, Masamichi ETO, Shoichiro HORIBA, Tomoaki TSUMURA, Hiroshi MATSUO: “Reducing Wasteful Recurrence of Aborts and Stalls in Hardware Transactional Memory”, Proc. 2013 High Performance Computing & Simulation Conf. (HPCS2013), Helsinki, Finland, pp.374-381 (Jul. 2013) [A.R.: 39%]
4. 橋本 高志良, 江藤 正通, 堀場 匠一朗, 津邑 公暁, 松尾啓志: “アドレス情報を利用した並列度の局所的な低減によるハードウェアトランザクショナルメモリの高速化”, 先進的計算基盤シンポジウム (SACSIS2013) 論文集, pp.162-169 (May. 2013)

報文

1. 山田 遼平, 橋本 高志良, 津邑 公暁: “待機アルゴリズムの改良によるトランザクショナルメモリの高速化”, 情処研報 (SWoPP2014), Vol.2014-ARC-211, No.1, pp.1-8 (Jul. 2014)
2. 橋本 高志良, 井出 源基, 山田 遼平, 堀場 匠一朗, 津邑 公暁: “共有変数に対する複合操作を排他実行するハードウェアトランザクショナルメモリの改良”, 情処研報 (ARC200), Vol.2014-ARC-208, No.22, pp.1-8 (Jan. 2014)
3. 山田 遼平, 堀場 匠一朗, 井出 源基, 橋本 高志良, 津邑 公暁: “投機的実行により並列度を向上させるハードウェアトランザクショナルメモリ ”, 情処研報 (ARC200), Vol.2014-ARC-208, No.23, pp.1-9 (Jan. 2014)
4. 橋本 高志良, 鈴木 大輝, 堀場 匠一朗, 津邑 公暁, 松尾 啓志: “ハードウェア・トランザクショナル・メモリにおけるトランザクション定義単純化モデルの検討”, 情

処研報 (SWoPP2013), Vol.2013-ARC-206, No.9, pp.1-8 (Jul. 2013)

5. 橋本 高志良，江藤 正通，鈴木 大輝，堀場 匠一朗，津邑 公暁，松尾 啓志 : “Read-after-Read アクセスを制御するハードウェアトランザクショナルメモリ”, 情処研報, Vol.2013-ARC-204, No.6, pp.1-8 (Mar. 2013)
6. 鈴木 大輝，橋本 高志良，堀場 匠一朗，江藤 正通，津邑 公暁，松尾 啓志 : “ハードウェアトランザクショナルメモリにおける競合パターンに応じた競合再発抑制手法の適用”, 情処研報, Vol.2013-ARC-204, No.5, pp.1-8（Mar. 2013）

参考文献

[1] Herlihy, M. and Moss, J. E. B.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. 20th Annual Int'l Symp. on Computer Architecture*, pp. 289–300 (1993).

[2] International Business Machines Corporation: *IBM System BlueGene Solution BlueGene/Q Application Development*, 1 edition (2012).

[3] Intel Corporation: *Intel Architecture Instruction Set Extensions Programming Reference, Chapter 8: Transactional Synchronization Extensions.* (2012).

[4] Shavit, N. and Touitou, D.: Software Transactional Memory, *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pp. 204–213 (1995).

[5] Sweazey, P. and Smith, A. J.: A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus, *Proc. 13th Annual Int'l. Symp. on Computer Architecture (ISCA'86)*, pp. 414–423 (1986).

[6] Censier, L. M. and Feautrier, P.: A New Solution to Coherence Problems in Multicache Systems, *IEEE Trans. on Computers*, Vol. C-27, No. 12, pp. 1112–1118 (1978).

[7] Rajwar, R. and Goodman, J. R.: Transactional Lock-Free Execution of Lock-Based Programs, *Proc. 10th Symp. on Architectural Support for Programming Languages and Operating Systems*, pp. 5–17 (2002).

[8] Bobba, J., Moore, K. E., Volos, H., Yen, L., Hill, M. D., Swift, M. M. and Wood, D. A.: Performance Pathologies in Hardware Transactional Memory, *Proc. 34th Annual Int'l Symp. on Computer Architecture (ISCA'07)*, pp. 81–91 (2007).

[9] Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D. and Wood., D. A.: Multi-

facet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset, *ACM SIGARCH Computer Architecture News*, Vol. 33, No. 4, pp. 92–99 (2005).

[10] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd Annual Int'l. Symp. on Computer Architecture (ISCA'95)*, pp. 24–36 (1995).

[11] Minh, C. C., Chung, J., Kozyrakis, C. and Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing, *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08)* (2008).

[12] Moravan, M. J., Bobba, J., Moore, K. E., Yen, L., Hill, M. D., Liblit, B., Swift, M. M. and Wood, D. A.: Supporting Nested Transactional Memory in LogTM, *Proc. 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 1–12 (2006).

[13] McDonald, A., Chung, J., Caristrom, B. D., Minh, C. C., Chafi, H., Kozyrakis, C. and Olukotun., K.: Architectural Semantics for Practical Transactional Memory, *ISCA '06 Proceedings of the 33rd annual international symposium on Computer Architecture*, pp. 53–65 (2006).

[14] Moss, E. and Hosking., T.: Nested Transactional Memory: Model and Preliminary Architecture Sketches., *In OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Langauges*, pp. 186–201 (2005).

[15] M, L., G, M. and A, G.: A Dynamically Adaptable Hardware Transactional Memory, *Microarchitecture(MICRO), 2010 43rd Annual IEEE/ACM*, pp. 27–38 (2010).

[16] Tomic, S., Perfumo, C., Kulkami, C., Armejach, A., Cristal, A., Unsal, O., Harris, T. and Valero., M.: Eazyhtm, Eager-lazy Hardware Transactional Memory, *In Procs. of the 42nd Symp on Microarchitecture (MICRO'09)*, pp. 145–155 (2009).

[17] Shriraman, A., Dwarkadas, S. and Scott., M. L.: Flexible Decoupled Transactional Memory Support, *ISCA '08 Proceedings of the 35rd annual international symposium on Computer Architecture*, pp. 139–150 (2008).

[18] Gaona, E., Titos, R., Acacio, M. E. and Fernández, J.: Dynamic Serialization Improving Energy Consumption in Eager-Eager Hardware Transactional Memory Systems, *Proc. Parallel, Distributed and Network-Based Processing 2012 20th Euromicro International Conference (PDP'12)*, pp. 221–228 (2012).

[19] Yoo, R. M. and Lee, H.-H. S.: Adaptive Transaction Scheduling for Transactional Memory Systems, *Proc. 20th Annual Symp. on Parallelism in Algorithms and Architectures (SPAA'08)*, pp. 169–178 (2008).

[20] Blake, G., Dreslinski, R. G. and Mudge, T.: Bloom Filter Guided Transaction Scheduling, *Proc. 17th International Conference on High-Performance Computer Architecture (HPCA-17 2011)*, pp. 75–86 (2011).

[21] Akpinar, E., Tomić, S., Cristal, A., Unsal, O. and Valero, M.: A Comprehensive Study of Conflict Resolution Policies in Hardware Transactional Memory, *Proc. 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'11)* (2011).

[22] Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D. and Wood, D. A.: LogTM: Log-based Transactional Memory, *Proc. 12th Int'l Symp. on High-Performance Computer Architecture*, pp. 254–265 (2006).

[23] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A. and Werner, B.: Simics: A Full System Simulation Platform, *Computer*, Vol. 35, No. 2, pp. 50–58 (2002).

[24] Alameldeen, A. R. and Wood, D. A.: Variability in Architectural Simulations of Multi-Threaded Workloads, *Proc. 9th Int'l Symp. on High-Performance Computer Architecture (HPCA'03)*, pp. 7–18 (2003).