卒業研究論文

競合の事前検出による
ハードウェアトランザクショナルメモリの高速化

指導教員　　津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成 23 年度入学　23115135 番

間下　恵介

平成 27 年 2 月 9 日

競合の事前検出による
ハードウェアトランザクショナルメモリの高速化

間下　恵介

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いる．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（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．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して更新がなされる際は，更新前あるいは更新後のデータ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，同一リソースに対する競合が発生していないかを検査する必要がある（競合検出）．トランザクショナルメモリのハードウェア実装であるハードウェアトランザクショナルメモリ（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，トランザクション操作のためのオーバヘッドを軽減している．

さて，HTM では一度競合したトランザクション同士が再度並列に実行される場合，競合が再発する可能性が高いことが事前に予測できるはずである．にもかかわらず，従来の HTM では過去の競合に基づく競合の発生予測を行わないため，一度競合したトランザクション同士で競合が頻発することがある．

そこで，本論文ではスレッドがトランザクションの実行を開始する際に競合の発生を予測し，競合が発生すると予測された場合，相手スレッドが実行を終了するまで待機することで競合の発生を事前に回避する手法を提案する．提案手法の有効性を検証するために，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．評価対象として GEMS microbench，SPLASH-2 および STAMP ベンチマークを用いた結果，8 スレッドで最大 54.3 %，平均 2.3 %，16 スレッドで最大 59.9 %，平均 17.7 %の実行サイクル数を削減した．

競合の事前検出による
ハードウェアトランザクショナルメモリの高速化

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，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り，また消費電力や配線遅延の相対的な増大により，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．

これらの流れを受け，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・コアが分担することで，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，ロック操作のオーバヘッド増大に伴う並列性の低下や，デッドロックの発生などの問題が起きる可能性がある．さらに，プログラムごとに適切なロック粒度を設定することは難しいため，ロックはプログラマにとって必ずしも利用しやすいものではない．

そこで，ロックを用い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，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適用したものであり，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することで，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，投機的に実行を進めることができ，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して更新がなされる際は，更新前あるいは更新後のデータ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，同一リソースに対する競合が発生していないかを常に検査する必要がある（競合検出）．トランザクショナルメモリのハードウェア実装である**ハードウェアトランザクショナルメモリ（Hardware Transactional Memory: HTM）**では，この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，トランザクション操作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，Intel のマイクロアーキテクチャHaswell[2] や IBM のスーパーコンピュータ BlueGene/Q[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，この HTM では，トランザクションは実行されるたびにほぼ同一の命令列を処理することが多く，毎回同じアドレスにアクセスするため，一度競合したトランザクション同士が再度並列に実行される場合，再び競合する可能性が高いことが事前に予測できるはずである．しかし，従来の HTM では過去の競合に基づいた競合の発生予測を行わず，一度競合したトランザクション同士が並列に実行されることで，競合が頻発することがある．その結果，ストールやアボートが何度も引き起こされ，HTM の性能を低下させてしまう．そこで本論文では，スレッドがトランザクションの実行を開始する際に競合の発生を予測し，競合が発生すると予測した場合，相手スレッドが実行を終了するまで待機することで，競合の発生を回避する手法を提案する．

以下，2 章では本研究の対象である TM および HTM の概要について説明し，3 章では本研究に関連する手法について説明する．4 章で HTM の問題点および提案手法について説明し，5 章でその動作モデルについて説明する．6 章で提案手法を評価し，7 章で結論を述べる．

2　TM の概要

本章では，本研究の対象となる TM および，それをハードウェアで実現したシステムである HTM について述べる．

2.1　トランザクショナルメモリ

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・コアが単一アドレス空間を共有する．したがって，異なるプロセッサ・コアによる同一メモリアドレスに対する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，ロックを用いてアクセス調停をする場合，デッドロックが発生する可能性がある．また，並列に実行するスレッド数や，使用するロック変数自体が増加した場合，ロックの獲得・解放操作に要するオーバヘッドも増加し，性能が低下する可能性がある．さらに，プログラムごとに適切なロックの粒度を設定することは困難である．例えば，粗粒度なロックを用いる場合，プログラムの構築は容易であるが，クリティカルセクションが大きくなるため並列性は損なわれる．一方，細粒度なロックを用いる場合，並列性は向上するが大規模なプログラムで

あるほど設計が困難となる．このように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．

そこでロックを用いない並行性制御機構として TM が提案されている．TM はデータベース上で行われ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手法である．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含む一連の命令列を，**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）:

並列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直列に実行した場合と同じであり，すべ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，処理の途中経過がほか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．しかし，複数のトランザクション内で同一アドレスへの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に**競合**として検出される．この操作のことを**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することで競合を回避する．これを**ストール（Stall）**という．また，複数のスレッドが互いのトランザクションをストールさせ合うとデッドロックに陥る可能性があるため，TM では一方のスレッドがトランザクションの途中までの実行結果を破棄することでデッドロックを回避する．この操作を**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のメモリ状態を復元し，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発生しなかった場合，トランザクション内で行われた変更を確定させる．この操作を**コミット（Commit）**という．なお，トランザクションを投機実行する場合，上述のとおり実行の途中結果が破棄される可能性があるため，更新されたデータと更新前のデータを保持しておく必要がある．そのため，TM ではトランザクション内で更新されたデータあるいは更新前のデータが，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．

TM は以上のように動作することで，競合が発生しない限りトランザクションを並列実行することができる．なお，TM で行われる競合検出やデータのバージョン管理などの操作はハードウェア上またはソフトウェア上に実装される．ソフトウェア上に TM を実装したソフトウェアトランザクショナルメモリ（STM）では，特別なハードウェア拡張の必要はないが，ソフトウェア処理のためのオーバヘッドが大きい．これに対し，ハードウェア上に実装された HTM では，競合を検出および解決する機構をハードウェアによってサポートしているため，STM に比べて速度性能が高い．そのため，本論文では速度性能の高い HTM を研究対象とする．

2.2 データのバージョン管理

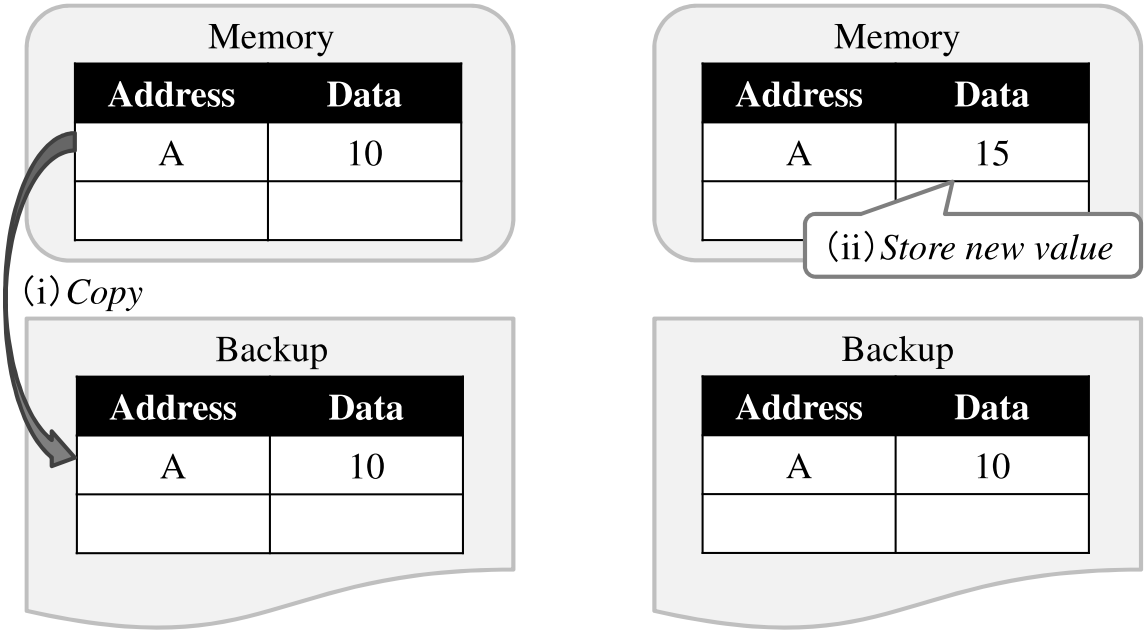
上述のとおり TM におけるトランザクションの投機実行では，実行結果が破棄される場合に備えてデータのバージョン管理を行う必要がある．このバージョン管理におけるデータの管理方式は以下の 2 つの方式に大別される．

Eager Version Management:
更新前のデータを別領域にバックアップし，更新後のデータをメモリに上書きする．コミットはバックアップを破棄するだけなので高速に行えるが，アボート時にバックアップされたデータをメモリに書き戻す必要がある．

Lazy Version Management:
更新前のデータをメモリに残し，更新後のデータを別領域にバックアップする．アボートは更新前のデータがメモリに残っているため高速に行うことができるが，コミット時に更新したデータをメモリに反映させる必要がある．

前者の Eager 方式は必ず実行されるコミットを高速に行い，必ずしも発生するとは限らないアボートに処理コストを払う方式である．そのため，アボートが頻発するようなプログラムでは処理が遅くなる可能性がある．しかし，Lazy 方式におけるコミットのオーバヘッドは削減の余地がほぼ無いのに対し，Eager 方式におけるアボートのオーバヘッドは，効率的なトランザクションスケジューリングによりアボートや競合の発生を抑制することで，削減できる余地が大いにある．よって本論文では 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** と**図 2** 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，メモリアドレス A に値が格納されている状態でトランザクションの実行



（a）データの退避　　（b）値の書き込み

図 1: Write アクセス時のバージョン管理

が開始されたとする．この状態からトランザクションの実行が進み，アドレス A に値 15 が書き込まれたとすると，図 1(a) に示す様に Write アクセスの対象アドレス A 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 (i)，図 1(b) のように書き込みの結果である 15 がメモリに上書きされる (ii).

次に図 1(b) の状態からさらに実行が進み，投機実行が成功した場合には，トランザクションがコミットされる．このとき，書き込みの結果である値 15 は既にメモリに保持されているため，図 2(a) に示すように，バックアップ領域の内容を破棄することでコミットを実現できる (iii).

これに対し，投機実行が失敗した場合にはトランザクションがアボートされる．このとき，図 2(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 (iv)．これにより，図 1(a) の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，アボート後トランザクションを再実行するためには，メモリ状態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に 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退避する．そして，アボート時にバックアップ領域を参照し，トランザクションの開始時点のレジスタ状態を復

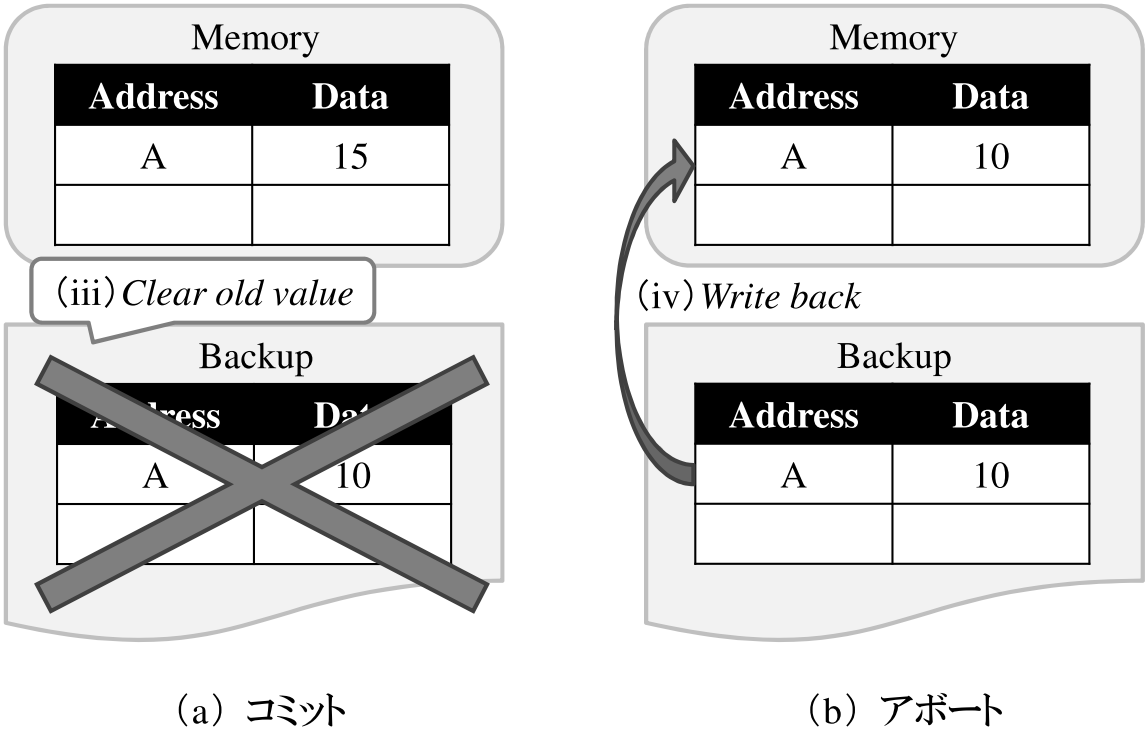


図 2: コミット，アボート時のバージョン管理

元する．

2.3 競合の検出と解決

トランザクションの Atomicity を保つためには，あるトランザクション内のメモリアドレスと他のトランザクション内のメモリアクセスとの間に競合が発生しているかどうかを検査する必要がある．そのため，トランザクション内でどのメモリアドレスがアクセスされたかを記憶しておかなければならない．これを実現するために，HTM では各キャッシュライン上に **Read/Write ビット**と呼ばれるフィールドが追加されている．トランザクション内で Read または Write アクセスが発生すると，アクセスされたキャッシュラインに対応する Read または Write ビットがセットされる．そして，各ビットは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 [4] の Illinois プロトコル [5] を拡張している．このプロトコルでは，あるスレッドがメモリアクセスする際，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にアクセスし

たスレッドに対し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，キャッシュラインの状態を変更する前に，当該ラインに追加されたReadおよびWriteビットを参照する．これにより，各スレッドがあるメモリアドレスにトランザクション内でアクセスしようとした際に，そのアドレスが他スレッドの実行するトランザクション内で既にアクセスされているかを検査することができる．この検査では，以下の3パターンのメモリアクセスを競合として検出する．

Read after Write（RaW）:
あるトランザクション内でWriteアクセスされたアドレスに対して，他のトランザクションからReadアクセスされるパターン．

Write after Read（WaR）:
あるトランザクション内でReadアクセスされたアドレスに対して，他のトランザクションからWriteアクセスされるパターン．

Write after Write（WaW）:
あるトランザクション内でWriteアクセスされたアドレスに対して，他のトランザクションからWriteアクセスされるパターン．

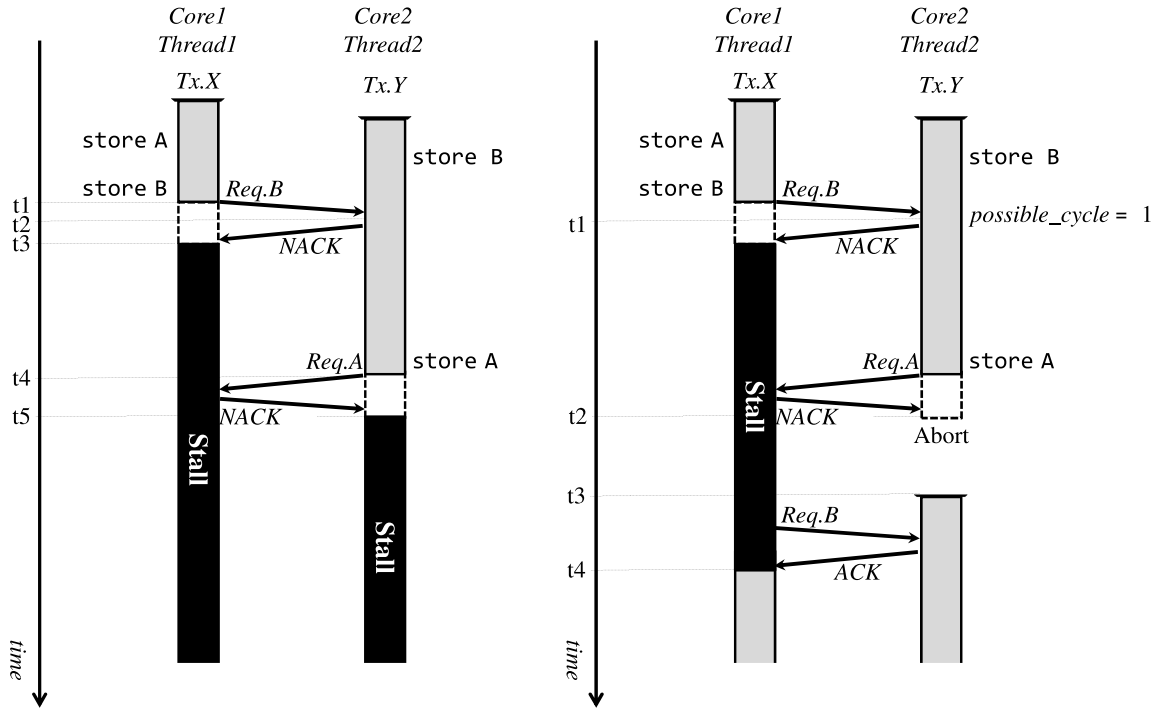
以上のようなアクセスパターンが検出されると，競合が発生したと判断し，競合を検出したスレッドからリクエストを送信したスレッドに対して*NACK*が返信される．これにより，*NACK*を受信したスレッドは自身のアクセスによって競合が発生したことを知ることができる．ここで，実際に*NACK*および*ACK*を送信するのはキャッシュディレクトリであるが，本論文では便宜的に，先行して当該アドレスにアクセスしたスレッドがこれらを送信するものとして説明する．

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存在するか否か検査する．

Lazy Conflict Detection:
トランザクションのコミットを試みた時点で，そのトランザクション内で行われた全てのアクセスに関して競合が存在するか否か検査する．

Lazy方式では，トランザクション内で競合が発生してから検出されるまでの時間がEager方式に比べて長くなり，無駄な処理が増大して実行効率が悪くなる．そのため，HTMでは競合検出方式としてEager方式が採用されている場合が多い．なお，一般的にEager方式では競合が発生した場合に競合相手のトランザクションがコミット，も



(a)デッドロックの発生　　(b)possible_cycleによるデッドロック回避

図 3: HTM におけるトランザクションの競合解決

しくはアボートされるまで実行トランザクションをストールさせる．そして，実行トランザクションをストールさせたスレッドは同じアドレスに対するリクエストを送信し続ける．その後，競合相手のスレッドが実行トランザクションを終了した場合，そのスレッドから *ACK* が返信されるため，実行トランザクションをストールさせていたスレッドは競合相手のトランザクションが終了したことを検知し，実行を再開できる．

しかし，**図 3**(a) に示すように，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るとデッドロックが発生する可能性がある．なお，この例では *Core1* と *Core2* という 2 つのコア上で 2 つのスレッド（*Thread1*，*Thread2*）が，それぞれトランザクション（*Tx.X*，*Tx.Y*）を並列に実行している．また，図中の縦軸は下に向かって時間が進むことを示しており，アクセス対象となっているアドレス A，B は共有メモリ上のアドレスであるとする．まず，*Thread1* が *Tx.X* の実行を開始した後，*Thread2* が *Tx.Y* の実行を開始しており，*Thread1* が `store A` を，*Thread2* が `store B` を実行済みである場合を考える．ここで，*Thread1* が `store B` を試みた場合，アドレス B に対するリクエストである *Req.B* を *Thread2* へ送信する（時刻 t1）．この *Req.B* を受信した *Thread2* は既に当該アドレス B に Write アクセス済みであるため，これは競合を引き起こすア

クセスパターンである WaW アクセスに該当する．したがって *Thread2* は競合の発生を検知し，*Thread1* へ *NACK* を返信する（t2）．*Thread1* は *NACK* を受信すると，自身の実行する *Tx.X* をストールさせる（t3）．なお，図中で省略しているが，*Thread1* はアドレス B に対するアクセス許可を受けるまで *Thread2* に対して定期的にリクエストを送信し続ける．この後，*Thread2* が `store A` を試みたとすると（t4），再び *Thread1* と *Thread2* の間で競合が発生するため，*Thread2* は自身の実行する *Tx.Y* をストールさせる（t5）．このように，複数のスレッドが自身の実行トランザクションをストールさせ，競合したトランザクションの終了を互いに待ちつづけると，デッドロック状態に陥ってしまう．

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[6] に倣った方法を採用している．具体的には図 3(a) のような状態に陥った時，トランザクションの開始時刻が競合相手のトランザクションよりも遅い場合に，そのトランザクションをアボートする．これは，各プロセッサ・コアに追加した *possible_cycle* と呼ばれるフラグを用いることで実現している．ここで，*possible_cycle* フラグを用いてデッドロックを回避する例を図 3(b) に示す．*Thread2* は *Thread1* へ *NACK* を返信する際，*Thread1* が自身よりも早くトランザクションを開始しているため，*possible_cycle* フラグをセットする（t1）．そして，*possible_cycle* フラグをセットしたスレッド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から *NACK* を受信した場合，デッドロックの発生を検知し，これを防ぐため自身の実行しているトランザクションをアボートする（t2）．このように *possible_cycle* フラグを利用することで，開始時刻の遅いトランザクションをアボート対象として選択する．そして，実行トランザクションをアボートした *Thread2* はトランザクション開始時点のメモリ状態，およびレジスタ状態を復元し，*Tx.Y* を再実行する（t3）．また，*Thread2* が実行トランザクションをアボートしたことで *Thread1* はアドレス B にアクセスできるようになるため，自身の実行トランザクションである *Tx.X* をストール状態から復帰させる（t4）．

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，再実行コストを抑える部分ロールバックに関する研究 [7, 8, 9] や，トランザクションの様々な情報に基づいて競合を抑制する研究 [10, 11, 12] など，HTM に関する数多くの研究が行われてきた．

既存の部分ロールバック手法を改良した Waliullah らの研究 [13] では，競合が引き起

こされたアドレスを記憶し，再度そのアドレスへアクセスする命令を実行する際，そのプログラムカウンタの値を利用して再実行開始位置を決定する．これにより，アボート時に発生する再実行のオーバヘッドを抑制している．しかし，競合しやすいアドレスに対するアクセスの直前にロールバックしてしまうことで，ロールバック後も競合が発生し，アボートが繰り返し引き起こされる恐れがある．

一方競合を抑制する Akpinar らの研究 [14] では，Eager 方式の HTM 向けに新しい競合解決ポリシーをいくつか提案している．それらのポリシーでは，ストールやアボートしたトランザクションの数やタイムスタンプなど，様々な情報に基づいてトランザクションの実行優先度が決定される．しかし，InactiveStall や FriendlyFire など [15]，一般的に HTM において性能低下を引き起こす競合パターンの発生状況の分析に重点を置いており，これらの問題に対するごく単純な解決手法しか提案していない．

このように，HTM の研究では様々な方面から改良が行われてきた．特にスレッドスケジューリングに関しては，その実行並列度に着目した改良手法が多く提案されてきた．武田ら [16] は，競合とトランザクション数の相関関係に着目し，動的にスレッド数を調整することで高速化を実現している．しかし，提案手法によって発生するオーバヘッドを評価しておらず，単一のベンチマークプログラムでのみ評価されているため，効果が明らかではない．また，橋本ら [17] は，特定のアクセスパターンにおける競合を抑制する手法を，山田ら [18] は，優先度を用いることで複数のスレッド間の競合を適切に処理することで，HTM の高速化を実現する手法を提案しており，これらは多くのベンチマークにおいて高い性能を実現している．

しかし，以上で述べた手法はいずれもトランザクションの実行中にスレッドの振る舞いを決定するため，スレッドがトランザクション実行前にある程度競合の発生を予測できる場合も，トランザクションの実行を開始することで，競合が発生してしまうことがある．そこで本論文では，スレッドがトランザクションの実行を開始する際に，そのトランザクションの実行によって他のスレッドと競合するかを予測し，競合の発生を抑制する手法を提案する．

4　競合の事前検出手法の提案

本章では，既存の HTM において性能低下を引き起こす問題について述べ，これを抑制する手法を提案する．

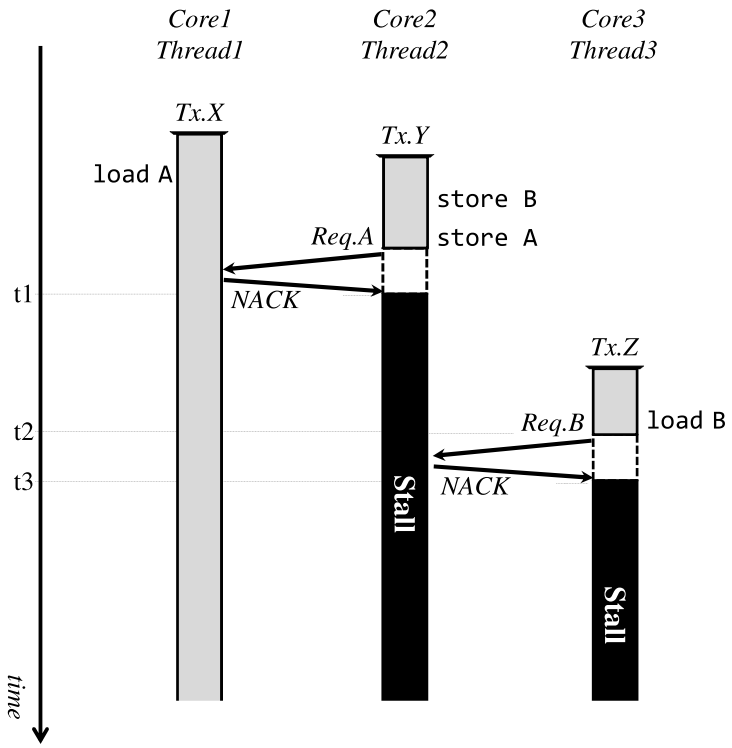


図 4: ストール中のトランザクションによる競合

4.1 既存の HTM における問題点

本節では，ストールおよびアボートにより HTM の性能が低下する原因について述べる．

4.1.1 ストール中のトランザクションによる競合の頻発

HTM では，競合の発生により *NACK* を受信したスレッドは，競合相手のスレッドがトランザクションをコミットまたはアボートするまで，自身の実行トランザクションをストールする．しかし，自身が実行するトランザクションのストールにより，トランザクションの実行時間が長くなることで，より多くの競合が引き起こされてしまう可能性がある．ここで，**図 4** に 3 つのスレッド上で並列に実行されるトランザクションの競合解決の様子を示す．この例では 3 つのスレッド（*Thread1*〜*Thread3*）がそれぞれ異なるトランザクション（*Tx.X*，*Tx.Y*，*Tx.Z*）を並列実行している．まず，*Thread1* が `load A` を実行した後，*Thread2* が `store B` を実行する．その後，*Thread2* が `store A` を試みると，アドレス A は *Thread1* によってアクセス済みであるため，*Thread1* は競合を検出する．そして *Thread2* に対して *NACK* を返信し，これを受信した *Thread2* は *Tx.Y* をストールする（t1）．その後，*Thread3* が *Tx.Z* の実行を開始して `load B` を試みたと

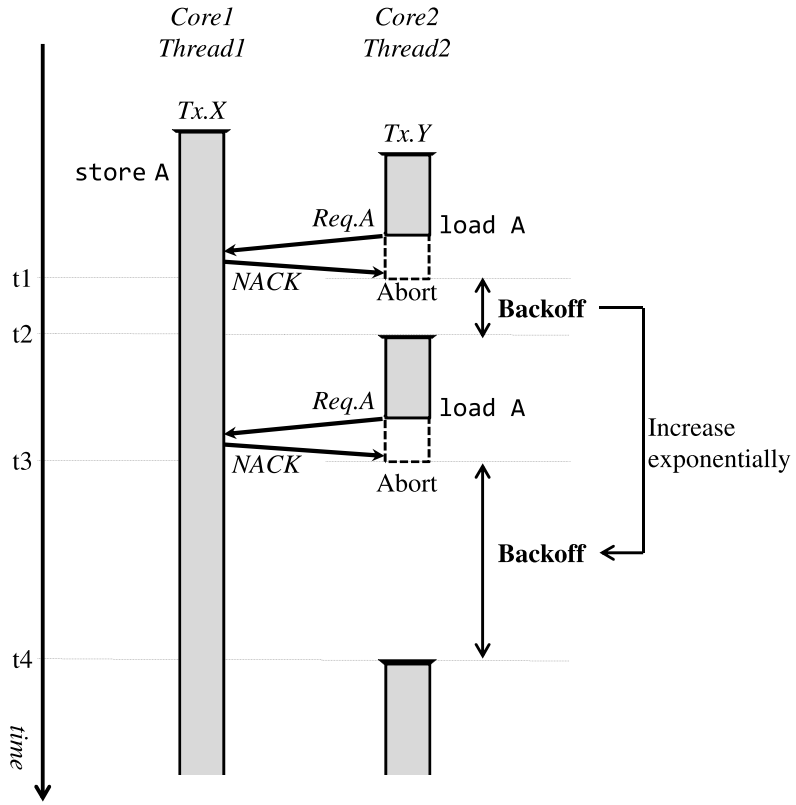


図 5: Backoff が指数関数的に増大する様子

すると（t2），アドレス B は *Thread2* によってアクセス済みであるため，*Thread2* は競合を検出する．そして *Thread3* に対して *NACK* を返信し，これを受信した *Thread3* は *Tx.Z* をストールする（t3）．しかし，*Thread3* が `load B` を試みる時刻 t2 よりも早く *Thread2* が *Tx.Y* をコミットしていれば，*Thread3* は競合せずに *Tx.Z* の実行を継続できたと考えられる．このように，ストールによってトランザクションの終了が遅れることで，競合がより多く発生する可能性がある．その結果 HTM の性能低下を引き起こす可能性がある．

4.1.2　アボートによるオーバヘッド

HTM ではアボート処理の後，即座にトランザクションを再実行してしまうと，再び競合が発生する可能性が高い．そこで，トランザクションがアボートされてから，再実行されるまで待機時間を設けることで競合の再発を防ぐ．この待機時間のことを **Backoff** という．なお，HTM ではこの Backoff による待機時間は **Exponential Backoff** というアルゴリズムを用いて算出されることが一般的である．この Exponential Backoff は，同一トランザクションが繰り返しアボートされることで Backoff を指数関数的に増大

させる．ここで，この Exponential Backoff により Backoff が増大する様子を**図 5** を用いて説明する．まず，*Thread1* が `store A` を実行した後 *Thread2* が `load A` を試みたとする．すると *Thread1* が競合を検出し，*Thread2* に *NACK* を返信する．一方，*NACK* を受信した *Thread2* は *Tx.Y* をアボートし，再実行を待機する（t1〜t2）．Backoff が終わると，*Thread2* は *Tx.Y* を再実行するが，*Tx.X* と再び競合が発生し *Thread2* は *Tx.Y* を再びアボートしてしまう（t3）．このとき *Thread2* が *Tx.Y* を再度アボートしたため，先ほど述べた Exponential Backoff に基づいて，Backoff が前回よりも長く設定される（t3〜t4）．なお，Backoff が長くなり過ぎることを防ぐために，事前に設定された回数だけアボートが繰り返された場合，それ以降アボートが繰り返されたとしても，Backoff は増大しない．また，トランザクションがコミットされる際，増大した Backoff はリセットされる．このように，Exponential Backoff を適用することで競合の頻発を防ぐことができるが，無駄な待機処理が発生することがある．

また，2.2 節で述べたようにアボート時には，トランザクション開始時点のメモリ状態およびレジスタ状態を復元するために，バックアップ領域に退避されたデータを元のメモリアドレス，およびレジスタに書き戻す必要がある．HTM ではこのバックアップ領域から元の状態を復元するコストが大きい．

以上で述べたように，スレッドのアボート時には，再実行までの待機時間と，アボート自体の処理に要する時間という 2 つのオーバヘッドが発生する．HTM ではこれらのオーバヘッドにより，性能が低下することがある．

4.2 競合の事前検出手法

本節では，前節で述べたようなストールおよびアボートによる HTM の性能低下を防ぐために，トランザクションの開始時に競合の発生を予測することで，競合を事前に検出する手法を提案する．

4.2.1 競合の予測方法

前節で述べたような問題は競合の頻発により発生する．特に，一度競合したトランザクション同士で何度も競合することが多い．この原因は，多くのトランザクション内の実行命令列は実行されるたびにほぼ同一であることから，トランザクションが実行されるたびに同じアドレスにアクセスするためである．

そこで，一度競合相手となったトランザクションが他のスレッドで実行されているか否かをスレッドがトランザクションの開始時に検査し，そのようなスレッドが存在する場合，競合が再発する可能性が高いと判断することで，一度競合したトランザク

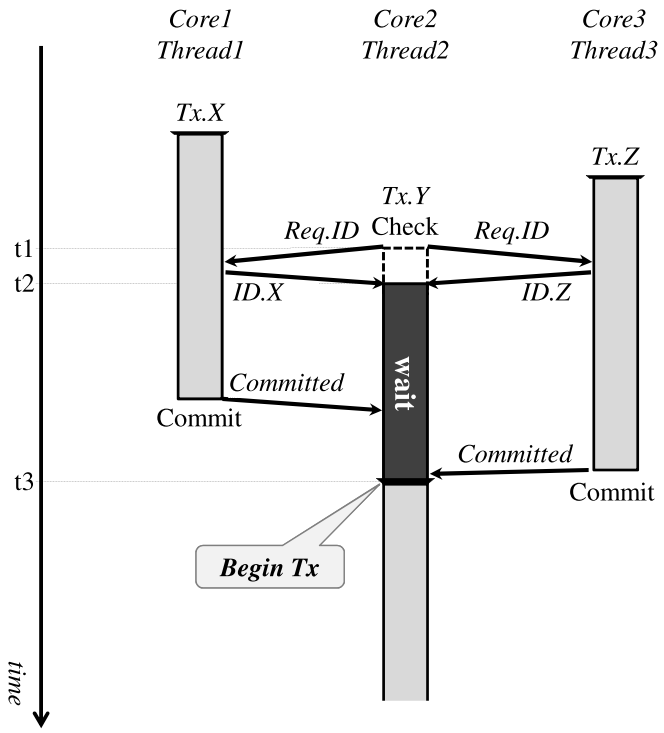


図 6: トランザクションの開始時の予測

ション同士による競合を予測する手法を提案する．これにより，競合を回避することで，トランザクションを実行中のスレッドは，競合することなく実行を継続でき，ストールやアボートの発生を防ぐことができる．

4.2.2 トランザクションの開始時の予測

前項で述べたように，トランザクションの実行中に競合が発生することを未然に防ぐため，スレッドがトランザクションを開始する際に競合の予測を行う．その動作例を**図 6** に示す．まず *Thread1* と *Thread3* が *Tx.X* と *Tx.Z* を実行中に，*Thread2* が *Tx.Y* の実行を開始したとする．このとき，*Thread2* は競合の予測のために，他の全てのスレッドで実行されているトランザクションの ID を問い合わせるリクエストを送信する（t1）．なお，この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．このリクエストを受信したスレッドは *Thread2* に対し自身の実行しているトランザクション ID を送信し，これを受信した *Thread2* は競合を予測する（t2）．このとき，*Tx.Y* が，*Tx.X* および *Tx.Z* と過去に競合したことがあるとすると，このまま実行を開始することで再度競合が発生する可能性が高い．そこで *Thread2* は競合の発生を予測し，競合相手のトランザクションである *Tx.X* および *Tx.Z* がコミットされるま

で実行を待機する (t2〜t3)．このように，各スレッドがトランザクションを開始する際，自身のトランザクションと過去に競合したことのあるトランザクションが他のスレッドで実行されている場合，実行の開始を待機することで競合を回避する．これに対し，競合相手となった *Thread1* および *Thread3* は自身の実行するトランザクションによって *Thread2* を待機状態にさせているため，自身のトランザクションをコミットする際に，*Thread2* に対しコミットしたことを伝える必要がある．そのため *Thread1* と *Thread3* はコミット時に，自身のコミットを伝える *Committed* メッセージを *Thread2* に送信する．なお，この *Committed* メッセージも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．*Thread2* は全ての競合相手スレッドから *Committed* メッセージを受信すると，全ての競合相手の実行が終了し，*Tx.Y* の実行を開始したとしても競合が発生しないと判断して待機を終了し，*Tx.Y* の実行を開始する．以上で述べたように動作させることで，トランザクションの開始時に競合を予測をする．その結果，競合が発生すると予測した場合，いずれのアドレスにもアクセスしていない状態で待機するため，ストールとは異なり，待機中のスレッドにより新たな競合が発生することを回避できる．

5　追加ハードウェアと動作モデル

本章では，競合の事前検出手法を実現するために拡張したハードウェアと，具体的な動作モデルについて説明する．

5.1　拡張したハードウェアの構成

トランザクション開始時に競合の発生を予測するために，既存の HTM を拡張して以下の 3 つのハードウェアを各コアに追加する．拡張したハードウェア構成を**図 7** に示す．なお，コア数を n とし，トランザクション ID の最大数を N とする．

競合相手記憶テーブル (E-Table):
各トランザクションごとに，競合相手のトランザクション ID を記憶するテーブル．

競合相手カウンタ (E-Counter):
競合の発生が予測された場合に，競合相手のトランザクションを実行しているスレッドの数を記憶するカウンタ．

待機スレッド記憶テーブル (W-Table):
他のスレッドを待機させているスレッドが，自身の待機させているスレッドの ID を記憶するテーブル．登録されている全てのスレッドに対して *Committed* メッセー

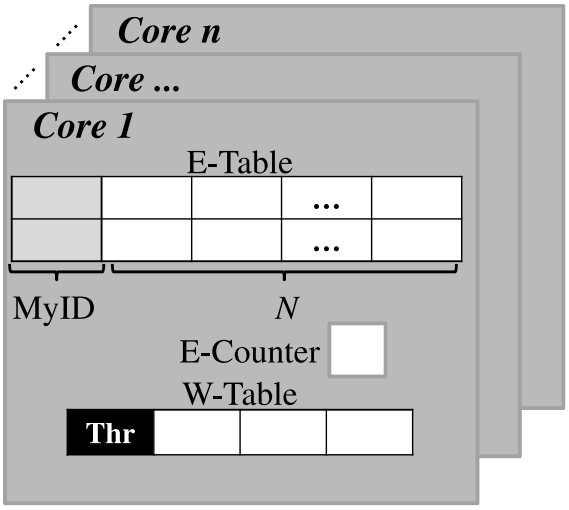


図 7: 拡張したハードウェアの構成

ジを送信すると，テーブルの値を全てクリアする．

提案手法では，過去に競合したことのあるトランザクションが他のスレッドによって実行されているか否か検査することで，競合の発生を予測するために，競合が発生するたびに競合相手のトランザクション ID を記憶する必要がある．そのため，各スレッドは自身の実行するトランザクションが競合すると，競合相手トランザクション ID を自身の競合相手記憶テーブルに登録する．そして，スレッドがトランザクションの実行を開始する際は，このテーブルを参照することで競合の予測をする．このとき，競合が発生すると予測したスレッドは，競合相手トランザクションを実行する全てのスレッドがコミットするまで実行を待機する必要がある．そのため，競合を検出したスレッドは，競合相手トランザクションを実行中のスレッド数を競合相手カウンタに記憶する．また，競合相手となった実行中のスレッドはコミットする際に，自身が待機させているスレッドにコミットを伝える必要がある．このコミットを伝える相手スレッドを待機スレッド記憶テーブルに記憶する．

5.2 初回競合時の動作

本節では，3 つのスレッド（*Thread1*〜*Thread3*）が，それぞれ異なるトランザクション *Tx.X*，*Tx.Y*，*Tx.Z* を並列に実行している**図 8** を例に，競合時の動作と競合相手記憶テーブルにトランザクション ID が登録される動作について述べる．まず，*Thread 2* により既にアクセスされているアドレス A に対して，*Thread1* が `store A` の実行を試みたとすると（t1），競合が発生するため，*Thread2* は *Thread1* に *NACK* を返信する．

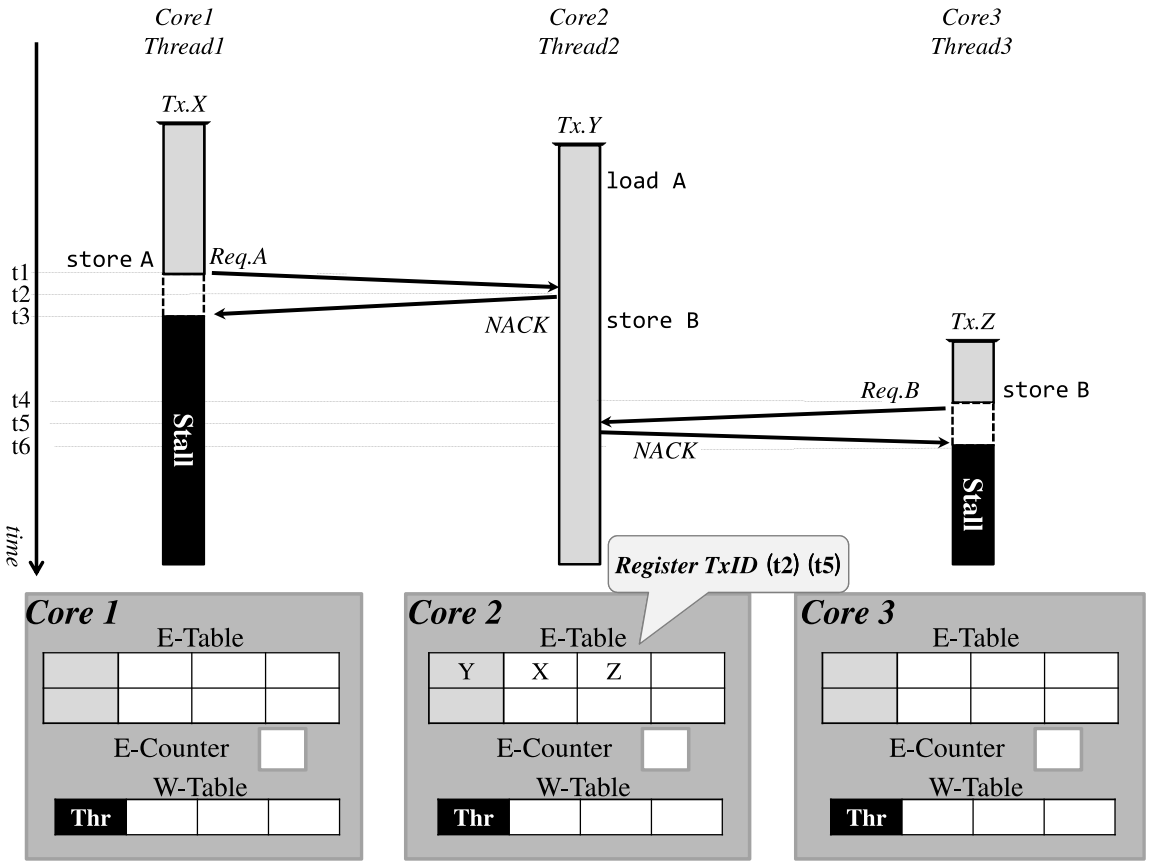


図 8: 初回競合時の動作と t6 における追加ハードウェアの様子

この時，*Thread2* は *Tx.Y* の競合相手として，*Tx.X* を自身の競合相手記憶テーブルに登録する（t2）．その後，*NACK* を受信した *Thread1* は *Tx.X* をストールする（t3）．同様に *Thread2* が `store B` を実行した後，*Thread3* が *Tx.Z* の実行を開始し，`store B` を試みたとする（t4）．この時，*Thread2* は競合を検出し，*Thread3* に対し *NACK* を返信する．同時に，*Thread2* は *Tx.Y* の競合相手として，*Tx.Z* を自身の競合相手記憶テーブルに登録する（t5）．一方，*Thread 3* は *NACK* を受信するため，*Tx.Z* をストールする（t6）．このように，初回競合時は競合相手記憶テーブルにトランザクション ID の登録をし，既存の HTM と同様の競合解決を行う．

5.3　トランザクションの開始時の動作

本節では，過去に競合が発生したトランザクションの開始時の動作について，5.1 節で述べた拡張ハードウェアを踏まえて説明する．

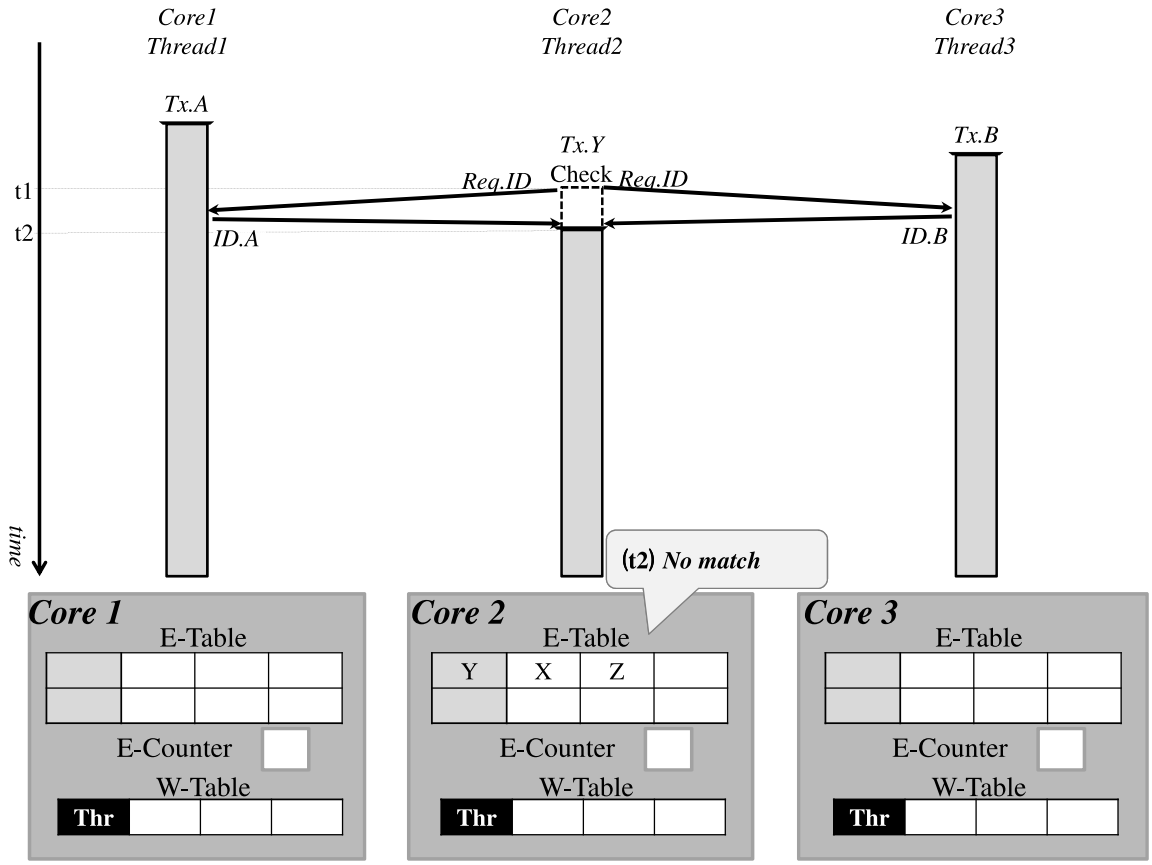


図 9: 競合が発生しないと予測した場合の動作と t2 における追加ハードウェアの様子

5.3.1 競合の予測

Thread1 と *Thread3* が *Tx.A* と *Tx.B* を並列実行している**図 9** を例に，トランザクションの開始時に，競合が発生しないと予測した場合の動作をまず述べる．この例では，*Tx.Y* の過去の競合相手として，*Tx.X* および *Tx.Z* が *Thread2* の競合相手記憶テーブルに登録されているものとする．*Thread2* が *Tx.Y* の実行を開始する際，競合の予測のため，他の全てのスレッドに対して，実行中のトランザクションの ID を問い合わせるリクエストを送信する（t1）．このリクエストを受信した *Thread1* および *Thread3* は，*Thread2* に対して自身の実行するトランザクションの ID である A および B を送信する．これを受信した *Thread2* は，受信した ID が競合相手記憶テーブルに登録されているかを確認することで競合の予測を行う．この例では，受信した A および B が競合相手記憶テーブルに登録されていないため，*Thread2* は競合が発生しないと予測し，*Tx.Y* の実行を開始する（t2）．

これに対し，図 9 の例と同様に *Thread2* が *Tx.Y* の実行の開始を試み，競合が発生す

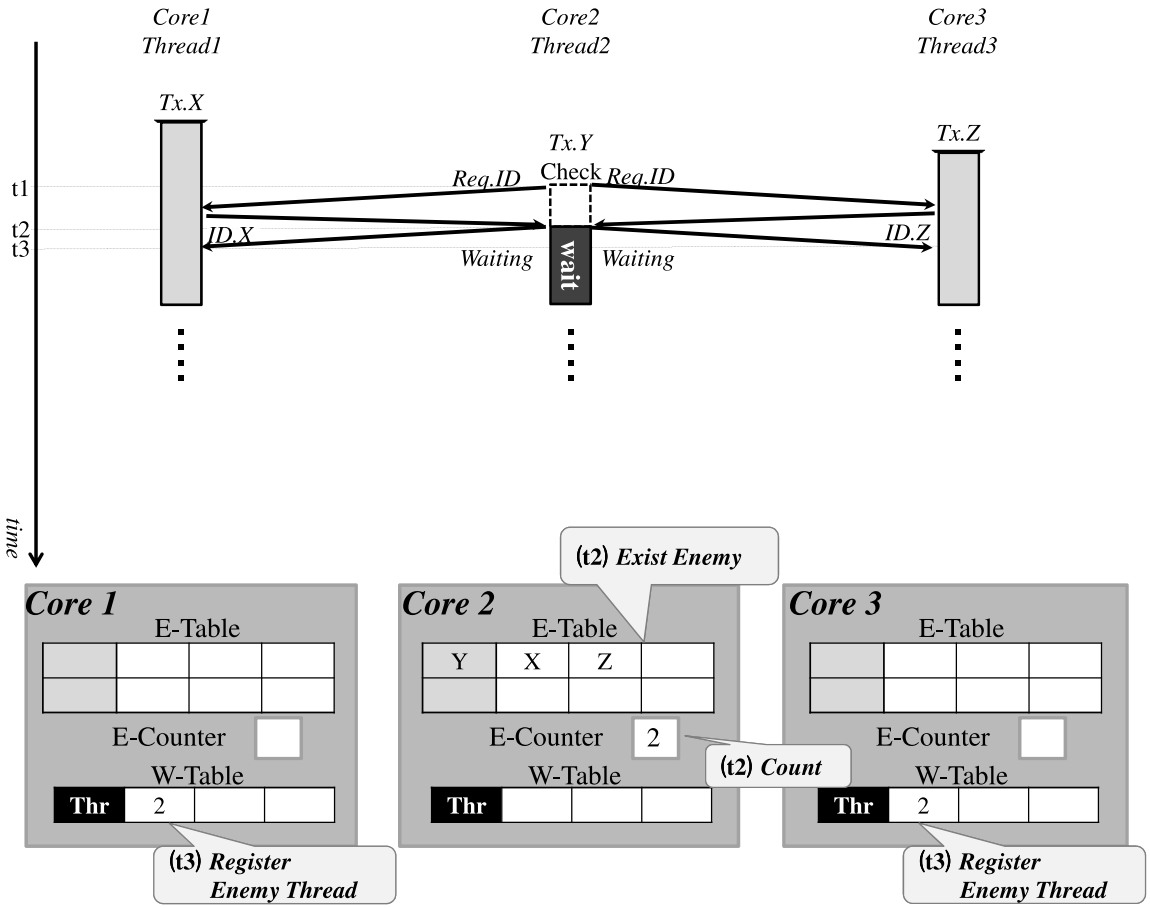


図 10: 競合が発生すると予測した場合の動作と t3 における追加ハードウェアの様子

ると予測した場合の動作を**図 10** に示す．この例も，競合相手記憶テーブルに図 9 と同様のトランザクション ID が登録されているものとする．*Thread1* と *Thread3* が *Tx.X* と *Tx.Z* を並列実行している中，*Thread2* が *Tx.Y* の実行を開始する際，競合の予測のため，他のスレッドに対して，実行中のトランザクションの ID を問い合わせるリクエストを送信する（t1）．これを受信した *Thread1* および *Thread3* は，*Thread2* に対して自身の実行するトランザクションの ID である X および Z を送信する．そして，これを受信した *Thread2* は競合の予測をする．この時，取得した ID である X および Z が *Tx.Y* の競合相手トランザクションの ID として競合相手記憶テーブルに登録されているため，*Thread2* は競合が発生すると予測し *Tx.Y* の実行を待機する（t2）．また，*Thread2* は競合相手カウンタに競合相手トランザクションを実行するスレッドの数である 2 を登録する．さらにこの時，*Thread1* および *Thread3* がそれぞれ実行している *Tx.X* および *Tx.Z* によって *Thread2* が実行を待機させられていることを伝えるために，*Thread2* 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した *Waiting* メッセー

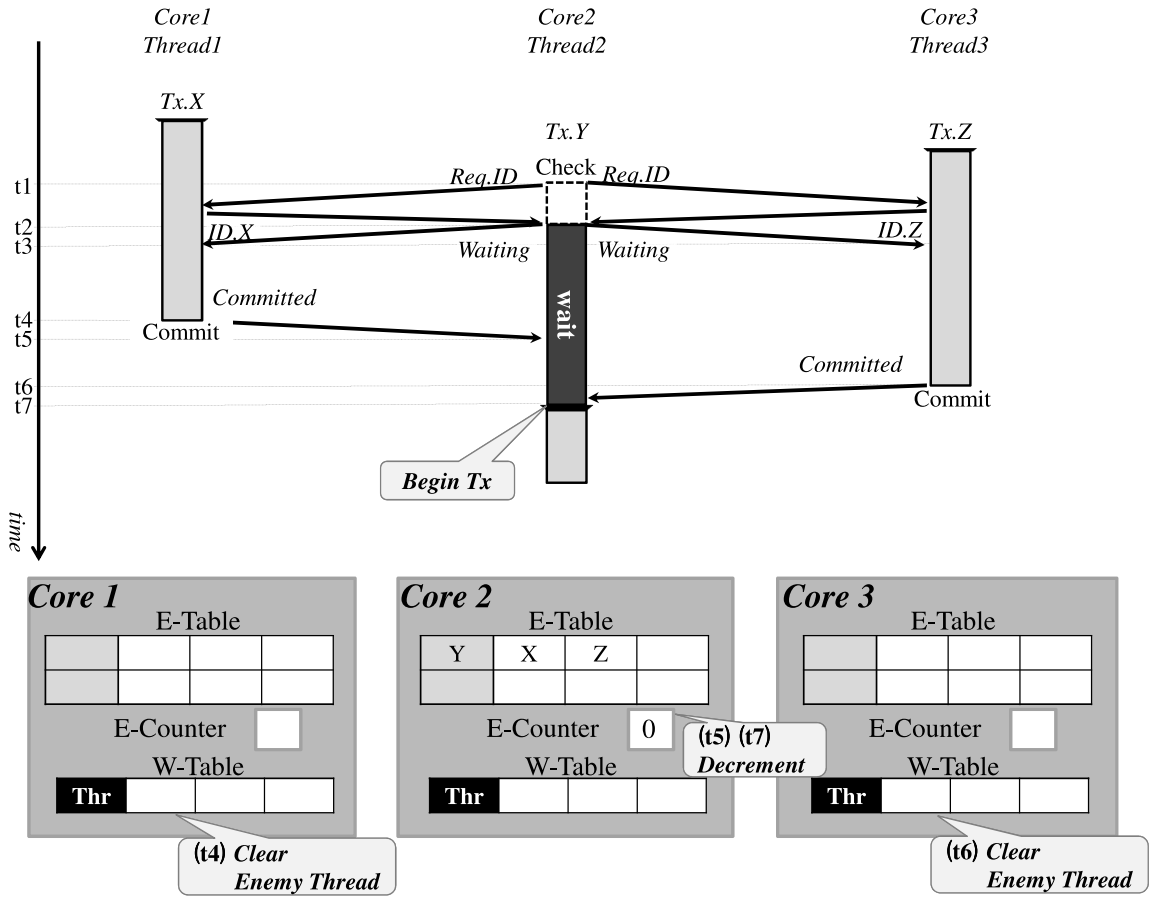


図 11: 競合相手がコミットする場合の動作と t6 における追加ハードウェアの様子

ジを *Thread1* および *Thread3* に対し送信する．一方，これを受信した *Thread1* および *Thread3* は，*Thread2* を待機させていることを記憶するために待機スレッド記憶テーブルに *Thread2* の ID である 2 を登録する（t3）．

5.3.2　競合相手スレッドのコミット時の動作

前項の図 10 で示した例において，実行を待機している *Thread2* が待機を終了し実行を開始するためには，競合相手のトランザクションがコミットされたことを知る必要がある．ここで，図 10 の t3 において *Thread2* が待機させられた後，*Thread2* が実行を開始する際の動作を表した**図 11** を用いて説明する．まず，前項で述べたように *Thread1* と *Thread2* は *Waiting* メッセージを受信したため，自身が待機させている *Thread2* の ID である 2 を待機スレッド記憶テーブルに登録する（t3）．この状態でその後 *Thread1* が *Tx.X* をコミットする際，待機スレッド記憶テーブルを参照し，自身が待機させている *Thread2* にコミットしたことを伝えるため，*Committed* メッセージを送信する．また，*Thread1* は待機させている全てのスレッドに *Committed* メッセージを送信し終わると，

待機スレッド記憶テーブルに登録したIDを全てクリアする（t4）．すると，*Thread1*から送信された*Committed*メッセージを受信した*Thread2*は，競合相手トランザクションを実行するスレッドのうちの1つが実行を終えたことを知り，競合相手カウンタの値をデクリメントする．その結果，競合相手カウンタの値は1となる（t5）．しかし，*Tx.Y*の競合相手である*Tx.Z*がまだ*Thread3*により実行されているため，*Thread2*は実行を待機し続ける．その後，*Thread1*と同様に*Thread3*がコミットする際も，待機スレッド記憶テーブルを参照し，自身が待機させている*Thread2*に対して*Committed*メッセージを送信する（t6）．そして，これを受信した*Thread2*は競合相手カウンタの値をデクリメントする．その結果，競合相手トランザクションを実行していたスレッドが全て実行を終えたため，競合相手カウンタの値が0となり，*Thread2*は実行を開始したとしても競合が発生しないと判断し，実行を開始する（t7）．以上のように動作することでトランザクションの実行開始時に競合を予測し，トランザクションの実行中に競合が発生することを防ぐ．

5.3.3 競合相手数に応じた動作

しかし，提案手法においてスレッドが競合の予測を誤り，本来必要のない待機をすることで性能が低下してしまう場合がある．ここで，既存のHTMと，提案手法を実装したHTMの動作を例に，提案手法において無駄な待機処理が発生してしまう様子を**図12**に示す．なお，図12(a)および図12(b)いずれの例においても*Tx.Y*は過去に*Tx.X*と競合したことがあり，図12(b)では*Thread2*の競合相手記憶テーブルに*Tx.Y*の競合相手として*Tx.X*が登録されているものとする．まず，一度競合したことがあるトランザクションが，既存のHTMにおいて並列実行可能な場合の例を図12(a)に示す．この例では2つのスレッド*Thread1*，*Thread2*がそれぞれ異なるトランザクション*Tx.X*，*Tx.Y*を並列実行している．まず，*Thread1*が`store A`を行った後，競合すること無く*Tx.X*をコミットする（t1）．その後，*Thread1*がアクセスしたアドレスAに対して*Thread2*が`store A`を試みた際，*Thread1*はトランザクションの処理を終了しているため，*Thread1*と*Thread2*の間で競合が発生しない．そのため，*Thread1*は*Thread2*に対して*ACK*を返信し，これを受信した*Thread2*は`store A`を実行し（t2），その後*Tx.Y*をコミットする．

一方，提案手法を適用し同様のトランザクションを同じタイミングで動作させる場合の例を図12(b)に示す．まず，*Thread1*が`store A`を行った後，*Thread2*が*Tx.Y*の実行を開始しようとする（t1）．このとき*Thread2*は競合の予測のため，実行しているトランザクションのIDを問い合わせるリクエストを全てのスレッドに送り，これを受

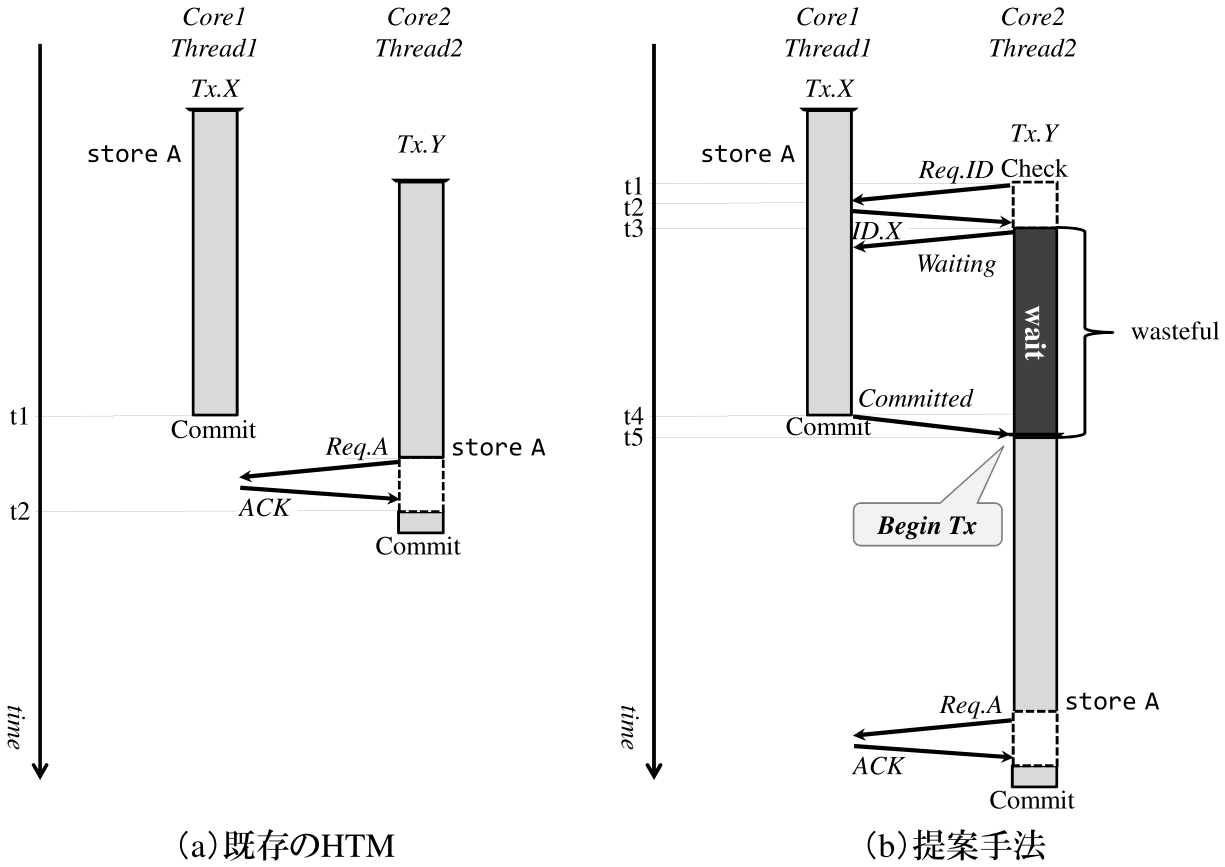


図 12: 無駄な待機が発生する様子

信した *Thread1* は自身の実行するトランザクションの ID である X を返信する（t2）．一方，ID を受信した *Thread2* は競合の予測を行う．その結果，受信した ID である X が競合相手記憶テーブルに登録されているため，競合が発生すると予測し，実行を待機するとともに *Waiting* メッセージを *Thread1* に対して送信する（t3）．これを受信した *Thread1* は *Tx.X* をコミットする際に *Thread2* に対し *Committed* メッセージを送信し（t4），これを受信した *Thread2* は *Tx.Y* の実行を開始する (t5).

以上で述べた 2 つの動作を比較すると，図 12(b) より，提案手法における待機処理が無駄となる場合があることが分かる．この無駄な待機処理が発生する原因は，提案手法によりスレッドが競合の発生を誤って予測してしまうためである．この問題は，競合が発生する可能性が高い場合のみ競合の発生を予測することで緩和できる．ここで，競合が発生する可能性は，トランザクションを並列に実行するスレッドの数に比例することが知られている．そのため提案手法では，待機を終了する際の競合相手カウンタの閾値を変更し，他のスレッドで実行されている競合相手トランザクションの数が

表 1: シミュレータ諸元

Processor	SPARC V9
#cores	32 cores
clock	4 GHz
issue width	single
issue order	in-order
non-memory IPC	1
D1 cache	32 KBytes
ways	4 ways
latency	3 cycles
D2 cache	8 MBytes
ways	8 ways
latency	20 cycles
Memory	4 GBytes
latency	450 cycles
Interconnect network latency	14 cycles

多い場合のみ待機することでこの問題の解決を図った.

6 評価結果と考察

前章で述べた拡張を HTM の一実装である LogTM[19] に実装し，評価を行った．本章ではその評価結果を示して考察する.

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[20] と GEMS 2.1.1[21] の組合せを用いた．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，また 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 1** 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は GEMS microbench，SPLASH-2[22]，および STAMP[23] から計 10 個を使用した．**表 2** に各ベンチマークプログラムの入力パラメタを示す．なお，各ベンチマークプログラムはそれぞれ 8，16 スレッド

表 2: 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
SPLASH2	
Barnes	512
Cholesky	tk14.0
Raytrace	teapot
STAMP	
Kmeans	random-n2048-d16-c16.txt
Vacation	-n8 -q10 -u80 -r65536 -t4096

で実行した.

6.2 評価結果

8 スレッドで実行した場合の評価結果を**図 13** および**表 3** に，16 スレッドで実行した場合の評価結果を**図 14** および**表 4** に示す．図中では，各ベンチマークプログラムの評価結果が，各 4 本のバーで表されている．これらのバーは左から順に，それぞれ

(**B**) 既存の LogTM（ベースライン）

(**E0**) 競合相手数 0 で実行を開始する提案モデル

(**E1**) 競合相手数 1 以下で実行を開始する提案モデル

(**E2**) 競合相手数 2 以下で実行を開始する提案モデル

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．5.3.3 項で述べた無駄な待機処理を削減するため，スレッドが待機を終了しトランザクションの実行を開始する際の競合相手カウンタの閾値を 0，1，2 と変化させて評価した．なお，フルシステムシミュレータ上でマルチスレッドを用いた動作のシミュレーションを行うには，性能のばらつきを考慮する必要がある [24]．したがって，各評価対象につき試行を 10 回繰り返し，得られた結果から 95%の信頼区間を求めた．信頼区間はグラフ中にエラーバーで示す.

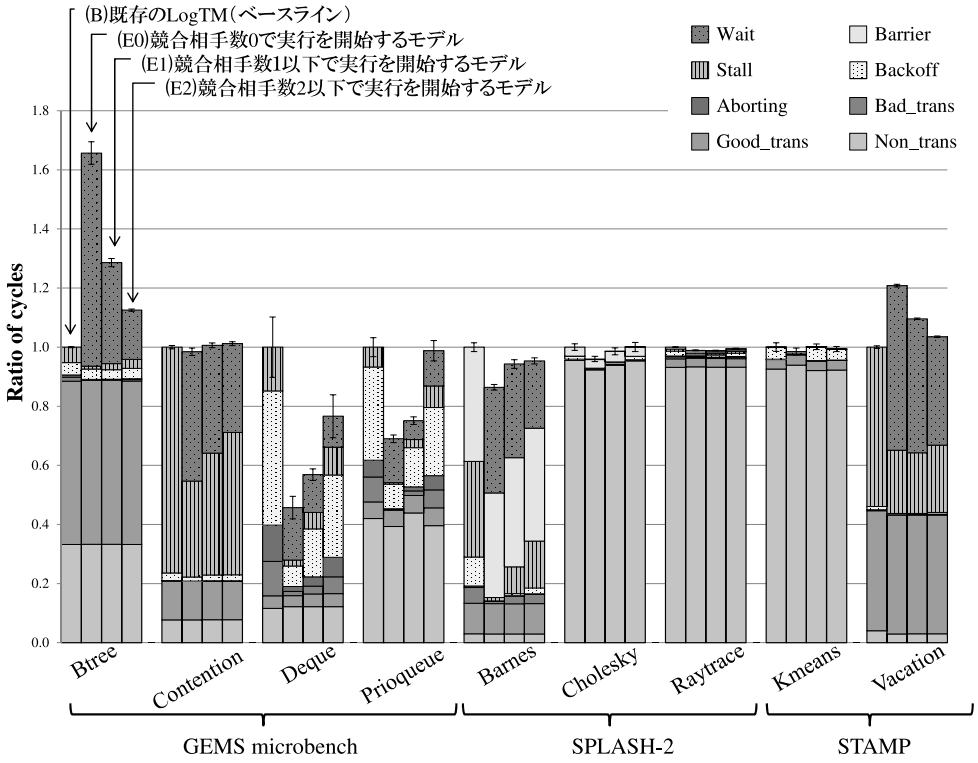


図 13: 8 スレッドでの各プログラムにおける実行サイクル数比

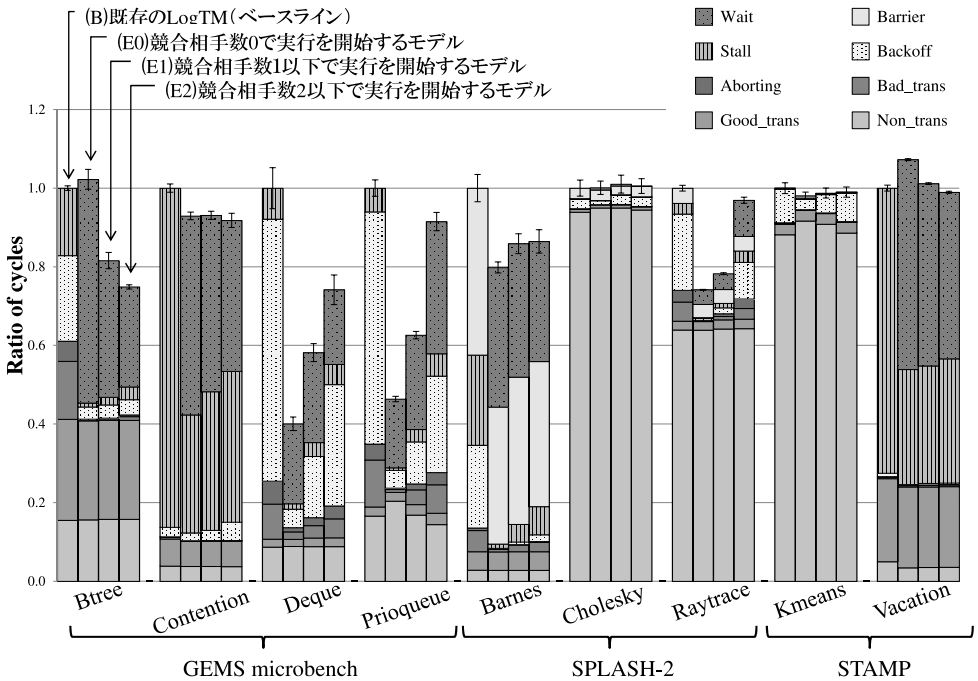


図 14: 16 スレッドでの各プログラムにおける実行サイクル数比

表 3: 8 スレッドでの各プログラムにおけるサイクル削減率

	GEMS microbench	SPLASH-2	STAMP	All
(E0) 平均	5.3%	6.2%	-9.7%	2.3%
最大	54.3%	13.6%	1.5%	54.3%
(E1) 平均	9.7%	2.7%	-4.9%	4.1%
最大	43.1%	5.7%	-0.1%	43.1%
(E2) 平均	2.7%	1.7%	-1.4%	1.5%
最大	23.4%	4.7%	0.6%	23.4%

表 4: 16 スレッドでの各プログラムにおけるサイクル削減率

	GEMS microbench	SPLASH-2	STAMP	All
(E0) 平均	29.6%	15.3%	-2.7%	17.7%
最大	59.9%	25.9%	1.9%	59.9%
(E1) 平均	26.1%	11.6%	0.1%	15.5%
最大	42.8%	21.7%	1.3%	42.8%
(E2) 平均	16.9%	5.4%	1.0%	9.5%
最大	25.9%	13.6%	1.0%	25.9%

また，図中の凡例はサイクル数の内訳を示しており，それぞれ以下のとおりである．

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: Backoff 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

評価の結果，提案手法により競合を抑制することで，全てのベンチマークにおいてストール，およびアボートに要したサイクル数を削減することができた．その結果，8 スレッドで Btree，Contention および Vacation をのぞく全てのベンチマークにおいて既存モデル (B) 以上の性能が得られ，16 スレッドで Btree および Vacation をのぞく全

てのベンチマークにおいて既存モデル (B) 以上の性能が得られた．また，提案モデル (E0)，(E1)，(E2) を比較すると，並列に実行できる競合相手トランザクションの数が増えるにつれて Wait サイクルが減少しているものの，ほとんどのベンチマークで性能が悪化している．これは，過去に競合したトランザクション同士が並列に実行され，競合が発生した回数が増加したためであると考えられる．さらに 8 スレッドと 16 スレッドの結果を比較すると，16 スレッドの方が大きく性能向上していることが分かる．これは，スレッド数の増加に伴い，既存モデル (B) では競合の発生回数が増加してしまったのに対し，提案手法では競合の発生を大幅に抑制できたためであると考えられる．

6.3 考察

本節では，各ベンチマークごとに詳細な考察を述べる．

GEMS microbench

まず，GEMS microbench の結果を見ると，(E0) において Deque，Prioqueue で (B) に対し大幅に実行サイクル数が減少している．これらのプログラムでは単一のトランザクションのみが繰り返し実行されており，(B) では複数のスレッドが同一のトランザクションを並列実行してしまうことで競合が頻発していた．しかし，提案手法により競合を大幅に抑制したことで Aborting，Stall，Backoff などのサイクル数が大幅に削減されたと考えられる．また，これらのプログラムではトランザクション内に含まれる実行命令数が少なく，競合相手トランザクションが比較的早くコミットされる．そのため，スレッドが競合相手トランザクションのコミットを待機したとしても，一回あたりの待機時間が短いため，Wait サイクルがあまり増大しなかったと考えられる．

次に，Contention では，提案手法で Stall サイクルが減ったものの，実行サイクル数に占める Stall の割合が多く，Wait サイクルも多いため，大きな性能向上は見られなかった．Contention も Deque，Prioqueue と同様に単一のトランザクションのみが繰り返し実行されており，(B) ではスレッドがトランザクションを並列実行することで競合が発生していた．これに対し提案手法では，競合の発生を防ぐことができたが，競合相手トランザクションのコミットにより，複数のスレッドが同時にトランザクションの実行を開始するため，結果的に競合が発生し，Stall が大きく減少しなかったと考えられる．また，Contention に含まれているトランザクションは実行命令数が非常に多く，コミットまでにかかるサイクル数が多いため，比較的 Wait サイクルが多くなったと考えられる．

一方，Btree はどの提案モデルにおいても (B) に対して大幅に Wait が増大してしまっている．その結果，8 スレッド，16 スレッドともに (E0) では性能が悪化した．この Btree には，2 種類のトランザクション（それぞれを仮に *Tx.R*，*Tx.W* とする）が存在している．*Tx.R* には load 命令のみが含まれ，*Tx.W* には load，store 命令の両方が含まれており，発生する競合のほとんどは *Tx.W* 同士で発生していた．しかし，*Tx.W* は分岐命令などにより，実行されるたびに異なるアドレスにアクセスすることで，並列に実行されたとしても競合が発生しない場合の方が多かった．そのため，提案手法では一度 *Tx.W* 同士で競合が発生すると，本来競合が発生しない場合も競合が発生すると予測してしまう．その結果，*Tx.W* は常に逐次実行されることになり，Wait サイクルが多くなったと考えられる．したがって今後，分岐命令を多く含むトランザクションを実行する際に，より適切に競合を予測する手法を考案していく必要がある．

SPLASH-2

SPLASH-2 ベンチマークでは各提案モデルにおいて，全てのプログラムで実行サイクル数が減少した．中でも Barnes では Stall や Backoff に要したサイクル数が大幅に削減できている．このプログラム中には 3 つのトランザクションが存在し，それらのトランザクション内で同一のアドレスがアクセスされることが多いため，(B) では競合が頻発していた．これを提案手法を適用することによって抑制でき，Stall や Backoff に要したサイクル数が削減されたと考えられる．しかし，Barnes に含まれるトランザクションは，全て同一のアドレスにアクセスするため，トランザクションが逐次実行されることで Wait サイクルが増大したと考えられる．

また，Cholesky においては (E0) で (B) に対して 5 %ほどの性能向上が見られた．他のプログラム同様，Stall や Backoff に要したサイクル数なども減少したが，中でも Non_trans に要したサイクル数が大きく減少した．これは，トランザクション外でのキャッシュアクセスによる False Sharing[25] を抑制できたためだと考えられる．なお，False Sharing とは，複数のスレッドが異なるデータにアクセスする際に，それらのデータが同一キャッシュライン上に存在していた場合に，誤って競合として検出されてしまうことである．(B) ではこの False Sharing により，トランザクション外の処理を行っているスレッドが NACK を受信することで不要なストールが発生し，その結果 Non_trans が増大してしまう．これに対し提案手法では，競合を抑制することでスレッドがトランザクションを処理する時間が短くなり，False Sharing の機会が減少したため，Non_trans に要したサイクル数が減少したと考えられる．

一方，Raytrace においてはどの提案モデルにおいても，8 スレッドでは (B) に対してほとんど性能向上は見られなかった．これは他のプログラムと同様 Stall や Backoff に要したサイクル数は大幅に削減されたが，実行時間の大半を占める Non_trans のサイクル数が変化しなかったため，このような結果になったと考えられる．しかし 16 スレッドでは，(B) において競合の発生回数が増加し，Backoff に要したサイクル数が多くなった．これを提案手法により抑制したことで，(E0) における 15%の実行サイクル数削減に繋がったと考えられる．

STAMP

STAMP ベンチマークでは，Kmeans で提案手法によって競合を抑制した結果，(B) と比較して Backoff のサイクル数が 10 分の 1 となり，実行サイクル数を削減することができた．しかし，Raytrace 同様，実行サイクル数の大半を Non_trans が占めており，全体としてあまり性能が向上しなかった．

一方，Vacation は提案手法により Stall に要したサイクル数を大幅に削減したが Wait サイクル数が増大し，結果として (B) に対して性能が悪化した．Vacation には非常に多くの実行命令数を含むトランザクション（仮に $Tx.S$ とする）が存在し，これが何度も実行されていた．この $Tx.S$ は分岐命令により，実行される度に異なるアドレスに対しアクセスされることが多いため，$Tx.S$ が並列実行されたとしても競合があまり発生しない．にも関わらず，提案手法を適用することで，一度競合が発生すると，それ以降 $Tx.S$ が逐次実行されるため，Wait サイクルが増大したと考えられる．これは Btree 同様，分岐命令を多く含むトランザクションを実行する際に，より適切に競合の発生を予測することで解決できると考えられる．

6.4 ハードウェアコスト

提案手法を実現するために追加したハードウェアのコストについて述べる．まず，競合相手記憶テーブルには，各トランザクション ID ごとに競合した相手のトランザクション ID を全て記憶できるだけのフィールドが必要である．そこで，全てのベンチマークを調査したところ，実行されるトランザクションの最大数は 5 であった．つまり，トランザクション ID は最大で 5 種類であるため，3bits でトランザクション ID を判別することができる．また，トランザクション 1 つあたりの競合相手トランザクションの最大数も 5 であるため，自身のトランザクション ID も含め，1 エントリあたり 6 つのフィールドが必要となり，エントリは深さが 5 行必要となる．つまり，競合相手

記憶テーブルは幅 3bits × 6 = 18bits，深さ 5 行の RAM で構成できる．また，競合相手カウンタには，自身を除いたスレッド数だけカウントできるだけのビット幅が必要であり，16 スレッドを並列実行する場合は最大で 4bits 必要となる．さらに，待機スレッド記憶テーブルには，自身が待機させているスレッド ID を登録できるだけのビット幅が必要である．したがって，16 スレッドを実行する場合，1 つのスレッドは最大で 15 個のスレッドを待機させるため，4bits × 15 = 60bits 必要となる．以上で述べた 3 つのハードウェアコストを合計すると，提案手法に必要なハードウェアコストは 1 コアあたり約 19Bytes となり，16 スレッドで実行可能な 16 コア構成のプロセッサの場合，19Bytes × 16core = 約 304Bytes となる．この 1 コアあたり約 19Bytes という数値は 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り，ごく小容量で提案手法を実現できる．

7　おわりに

本論文では，過去に競合したトランザクション同士で競合が頻発していることに着目し，過去に競合した相手トランザクション ID を記憶し，これを用いてトランザクションの開始時に競合の発生を予測することで競合を事前に検出する手法を提案した．各スレッドは，競合が発生すると予測した場合，競合相手トランザクションがコミットされるまで，自身のトランザクションの実行を待機することで競合の発生を回避する．提案手法の有効性を確認するために GEMS microbench，SPLASH-2 および STAMP ベンチマークプログラムを用いて評価を行った結果，既存の HTM と比較して 8 スレッドで最大 54.3 %，平均 2.3 %，16 スレッドで最大 59.9 %，平均 17.7 %の実行サイクル数が削減されることを確認した．提案手法によって性能向上が得られた要因として，既存の HTM と比較して競合の発生回数を抑制したことによりアボートの発生回数を大幅に削減したことが挙げられる．これにより Backoff が削減され，多くのベンチマークにおいて性能が向上した．また，既存の HTM に対して提案手法を適用する上で必要となるハードウェアのコストを概算したところ，16 コア構成のプロセッサの場合約 304Bytes となり，小容量のハードウェアを追加することで実現できることを示した．

しかし，分岐命令などにより処理される実行命令列が変わるようなトランザクションにおいて，競合が発生しない場合も提案手法により競合が発生すると予測し，無駄な待機をする場合があった．また，実行命令数を多く含むトランザクションの処理を待機すると待機時間が長く，性能が低下することがあった．したがって今後，これらの原因について詳細に調査し，無駄な待機時間を削減する方法の考案と，より適切な

競合の検出方法について検討していきたい.

謝辞

本研究のために，時に厳しく，時に優しく御指導を賜わった津邑公暁准教授に深く感謝します．私のような未熟者に，様々な機会を与えてくださり，成長させていただいたからこそ，この論文が完成したのだと思っています．また，多くの助言，協力をいただいた松尾啓志教授，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．また，1 年間ともに研究してくださった津邑研究室，松尾研究室，齋藤研究室，および松井研究室の方々に深く感謝致します．特に津邑研究室の方々には大変お世話になりました．まず，隣の席で些細な悩みを聞いてくださったり，僕を優しく見守ってくださった竹嶌良氏に深く感謝致します．また，同じ研究テーマで研究の進め方など，多くのことをご指導していただいた山田遼平氏に深く感謝致します．そして，様々な刺激を与えてくださった濱西夏生氏に深く感謝致します．あなたとともに夜遅くまで研究を進め，食事をともにしながらいろいろなことを語りあえた思い出は，とてもかけがえのないものになりました．そして，橋本高志良氏に深く感謝致します．先輩には研究について多くのご助言をいただいただけでなく，研究の楽しさや喜び，そして無限の可能性を感じさせていただきました．最後となりますが，私の研究生活に関わっていただいた全ての方々に深く感謝致します.

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

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

[4] 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).

[5] 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).

[6] 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).

[7] 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).

[8] 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).

[9] 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).

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

[11] 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).

[12] 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).

[13] Waliullah, M. M. and Stenstrom, P.: Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems, *Proc. 22nd IEEE Int'l Symp. on Parallel and Distributed Processing (IPDPS 2008)*, pp. 1–11 (2008).

[14] 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).

[15] 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).

[16] 武田進, 島崎慶太, 井上弘士, 村上和彰: トランザクショナルメモリにおける並列実行トランザクション数動的制御法の提案とその評価, 信学技報, Vol. 108, No. ICD-28, pp. 81–86 (2008).

[17] 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 (ACS44), Vol. 6, No. 4, pp. 58–71 (2013).

[18] Yamada, R., Hashimoto, K. and Tsumura, T.: Priority-Based Conflict Resolution for Hardware Transactional Memory, *Proc.2nd Int'l Workshop on Computer Systems and Architectures(CSA'14),held in conjunction with CANDAR'14*, pp. 433–439 (2014).

[19] 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).

[20] 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).

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

[22] 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).

[23] 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).

[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).

[25] Horiba, S., Asai, H., Eto, M., Tsumura, T. and Matsuo, H.: Fine-Grain Conflict

Management for Hardware Transactional Memory Systems Employing Eager Version Management, *Proc 4th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures(PARMA2013),held in conjunction with HiPEAC'13*, p. 6 (2013).