修士論文

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

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 23 年度入学　23413514 番

江藤 正通

平成 25 年 2 月 5 日

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

江藤 正通

内容梗概

マルチコア環境におけるスレッドレベル並列性を活用した並列プログラミングでは，共有リソースへのアクセスの調停にロックが広く利用されている．しかし，ロックを用いた場合には，デッドロックの発生や並列性の低下等の問題がある．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 (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をメモリアクセスに応用したものであり，複数トランザクションによる同一アドレスへのアクセスを監視することで競合を検出する．この TM をハードウェア上に実現したものは HTM と呼ばれ，HTM は一般的に possible_cycle と呼ばれるフラグを使ったアボート対象の選択方法を採用している．

しかしこの方法では，スレッドスケジューリングの効率が悪く，トランザクションのアボートやストールが頻発する場合がある．たとえば，あるアドレスに対する書き込みを実行するトランザクションに対しそのアドレスから値を読み出すトランザクションが複数存在する場合，書き込み側のトランザクションが長期間待機させられ，読み出し側のトランザクションがアボートを繰り返してしまう可能性がある．また，競合を頻繁に引き起こす可能性のあるトランザクションが並列に実行される場合，結果的にはスケジューリングの向上に一切寄与しないようなストールが発生してしまう可能性もある．

そこで本論文では，このような状況で発生するアボートやストールを抑制する手法を 2 つ提案する．1 つ目は，書き込み処理を実行できず飢餓状態に陥ってしまったトランザクションを優先的にコミットさせる手法である．2 つ目は，競合が頻発するトランザクションを逐次的に実行する手法である．これらの手法により，適切なスレッドスケジューリングを実現する．

提案した手法の有効性を検証するために，既存の LogTM に提案手法をそれぞれ実装し，GEMS 付属の microbench，SPLASH-2，STAMP の 3 種のベンチマークを用いてシミュレーションによる評価を行った．その結果，既存の LogTM に比べて，提案した 2 つの手法を組み合わせたモデルでは，最大 72.2%，平均 28.4%の実行サイクル数

を削減できることを確認した.

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

目次

1 はじめに

これまでのプロセッサ高速化技術は，スーパスカラに代表されるような（Instruction Level Parallelism: ILP）に基づくさまざまな手法を中心としつつ，集積回路の微細化による高クロック化を半導体技術の向上により実現することで支えられてきた．しかしながらプログラム中のILPには限界があり，また消費電力や配線遅延の相対的増大により，クロック向上も頭打ちになりつつある．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集積したマルチコア・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，今までひとつのコアが担っていた仕事を複数のプロセッサ・コアで分担することで，単一コアでの実行よりもスループットを向上させることができる．たとえば，スレッド並列性を利用して，プログラムを並列に実行することで，実行時間の短縮が期待できる．

このようなマルチコア環境では，複数のプロセッサ・コア間で単一アドレス空間を共有した共有メモリ型の並列プログラミングモデルが広く利用される．そのようなプログラミングモデルでは共有リソースへのアクセスを調停する必要があり，その制御を行う機構として一般的にはロックが用いられている．しかし，ロックを用いたプログラミングでは，デッドロックの発生を考慮し，また各プログラムで適切なロックの粒度を設定しなければ並列性を向上させるのは困難である．そのため，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**，以下**TM**）[1]が提案されている．TMは，トランザクションとして定義された命令列を投機的に実行することで，ロックを用いた場合と比較して速度性能が向上する．

このような利点からTMは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Blue gene/Q[3]には，このTMをハードウェアで実現した**ハードウェア・トランザクショナル・メモリ**（**Hardware Transactional Memory**，以下**HTM**）が実装されている．一般的なHTMでは，各キャッシュラインに対してreadビットとwriteビットという，トランザクション内で発生したRead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，キャッシュコヒーレンスリクエストを受け取ったときにこれらのビットを参照することで，そのキャッシュラインで競合が発生したかどうかを判定する．競合が発生した場合，その競合を引き起こしたトランザクションを実行するスレッドは，競合相手のトランザクションが終了するまで実行を一時的に停止させる．これをストールという．ここで，

トランザクションを実行する複数のスレッドでストールが発生した場合，それらのスレッドがお互いのトランザクションが終了するまで待つ，デッドロック状態に陥る可能性がある．これを回避するために，デッドロックに関わるどれか 1 つのトランザクションの途中結果を全て破棄するアボート操作が行われる．その後，そのトランザクションの実行を開始した時点のメモリおよびレジスタ状態が復元され，アボートされたトランザクションは再実行される．

一般的な HTM では，possible_cycle と呼ばれるフラグを用いることでアボート対象が決定されるが，この方法ではスレッドスケジューリングの効率が悪く，トランザクションのストールやアボートが頻発する場合がある．たとえば，あるアドレスから値を読み出す複数のトランザクション (reader) と，そのアドレスに対する書き込みを実行するトランザクション (writer) が存在する場合，reader がアボートを繰り返し，writer は長期間待機させせられてしまうことがある．このような writer は starving writer と呼ばれ，性能に悪影響を及ぼす可能性がある．

また，競合を頻繁に引き起こす可能性のあるトランザクションが並列に実行される場合，開始時刻の遅いトランザクションによって開始時刻の早いトランザクションがストールさせられる状況が発生しやすい．このストール中に，これらのトランザクション間で再び競合が発生した場合，開始時刻の遅いトランザクションがアボートされる．このアボートにより，結果的には開始時刻の早いトランザクションをストールさせる必要がなかったことになる．このようなストールは futile stall と呼ばれ，starving writer が発生する場合と同様に，性能に悪影響を及ぼすと考えられる．

そこで本論文は，このような状況で観測される競合のパターンに着目し，これらの競合を抑制する手法をそれぞれ提案する．まず 1 つ目は，starving writer が発生した場合に reader を待機させることで，starving writer を優先的にコミットする手法である．これにより，starving writer のストールおよび reader のアボートの繰り返しが抑制される．2 つ目は，競合が頻繁に引き起こされるトランザクションを逐次的に実行する手法である．これにより，それらのトランザクションを並列実行した場合に発生するアボートを防ぐことができ，futile stall も抑制できると考えられる．

以下，２章では本研究の背景である TM および HTM の概要について説明する．3 章および 4 章では，競合パターンが発生する状況と，それを解決する手法およびその実装方法について述べる．5 章でこれらの手法を評価し，6 章で関連研究について述べる．最後に 7 章で本論文全体をまとめる．

2　研究背景

本章では，TM の基本概念および TM をハードウェアで実現する HTM について説明する．

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，共有メモリ型のプログラミングモデルが広く利用されており，そのようなプログラミングモデルでは，複数のプロセッサ・コアが単一アドレス空間を共有する．このため，同一のメモリアドレスに対して，複数のプロセッサ・コアからのアクセスが発生した場合，結果の一貫性を保つために共有リソースへのアクセスを調停する必要がある．そのような操作を行う機構として，一般的にはロックが用いられている．しかし，ロックを用いた制御ではデッドロックが発生する危険性がある．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が増加し，性能が低下する可能性もある．さらに，大規模で複雑なプログラムであるほど，適切なロックの粒度を設定することは困難である．たとえば，粗粒度ロックを用いる場合では，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．一方で，細粒度ロックを用いる場合では，プログラムの並列性は向上するがプログラムの設計が難しい．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である 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をメモリアクセスに対して適用した手法である． TM では，クリティカルセクションを含む一連の命令列が，以下の 2 つの性質を満たす**トランザクション**として定義される．

シリアライザビリティ(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直列すなわち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．

アトミシティ(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 は各トランザクション内でアクセスされるメモリアドレスを常に監視し，比較する．ここで，複数のトランザクション間で同一アドレスへのアクセスが発生した場合，少なくとも一方が Write アクセスならば，これらのアクセスはトランザクションの性質を満足しないため**競合**として判定される．この操作を**競合検出**（**Conflict Detection**）という．もし競合が発生したと判定された場合は，片方のトランザクションの実行を中断する．これを**ストール**という．さらに，複数のトランザクションがストールしている状態で，デッドロックの可能性があると判断された場合，デッドロックに関わるどれか 1 つのトランザクションの途中結果を全て破棄する．これを**アボート**という．そしてアボートされたトランザクションの開始時点の状態を復元し，トランザクションを再実行する．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実行された全ての結果をメモリに反映させる．これを**コミット**という．なお，アボートおよびコミット操作を行うためには，更新される前の古い値と更新した値の両方を保持しておく必要がある．このため，これらの値はお互いに干渉することはない別々の領域に保持される．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並列に実行することができる．これによりロックを適用した場合よりもプログラムの並行性が向上するため，コア数に応じて性能がスケールすることが期待できる．また，プログラマはロックの粒度を考慮する必要がなくなるため，容易に並列プログラムを構築することができる．

ここで，TM で行われる競合の検出やコミットおよびアボート等の操作は，ハードウェア上またはソフトウェア上に実装されることで実現される．これらのうち，ハードウェア上に実装された TM は HTM と呼ばれる．一般的に HTM は，トランザクション内で更新した値と更新前の古い値とを併存させるために，片方をキャッシュ上に保持し，もう片方を別の表に退避している．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ってサポートしている．一方で，ソフトウェア上に実装された TM はソフトウェア・トランザクショナル・メモリ（STM）[4] と呼ばれる．STM は，HTM のような特別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って実現されるため，オーバヘッドが大きい．したがって，HTM は STM に比べて速度性能が高い．本論文では，この速度性能の高い 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的実行では，実行結果が破棄される可能性があるため，トランザクション内で更新した値と更新前の値と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発生したWriteアクセスにより更新したデータ，あるいは更新される前の古いデータを，そのアドレスとともに別の領域に保持する．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バックアップし，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を破棄するだけなので高速に行えるが，アボート時にはバックアップ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登録する．アボートは高速に行えるが，コミット時にメモリへの値のコピーが必要となる．

前者の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が，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．よって本論文では，eager方式を研究の対象とする．

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． これらの図にあるMemoryおよびBackupはそれぞれメモリ，バックアップ領域を表している．まずメモリアドレス`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．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が書き込まれるとすると，まず図1(a)に示すように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），その後図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，投機的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，バックアップ領域の内容を破棄するだけでコミットを実現できる．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．アボート時は，図2(a)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，図2(b)のようにトランザクション開始時点のメモ

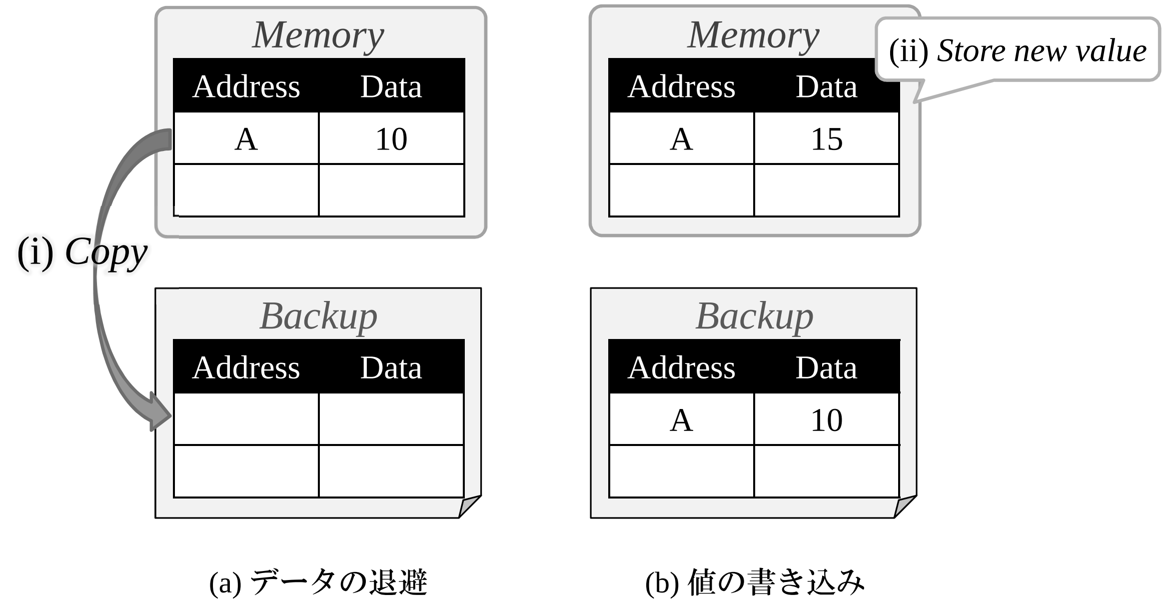


図 1: Store 実行時のバージョン管理

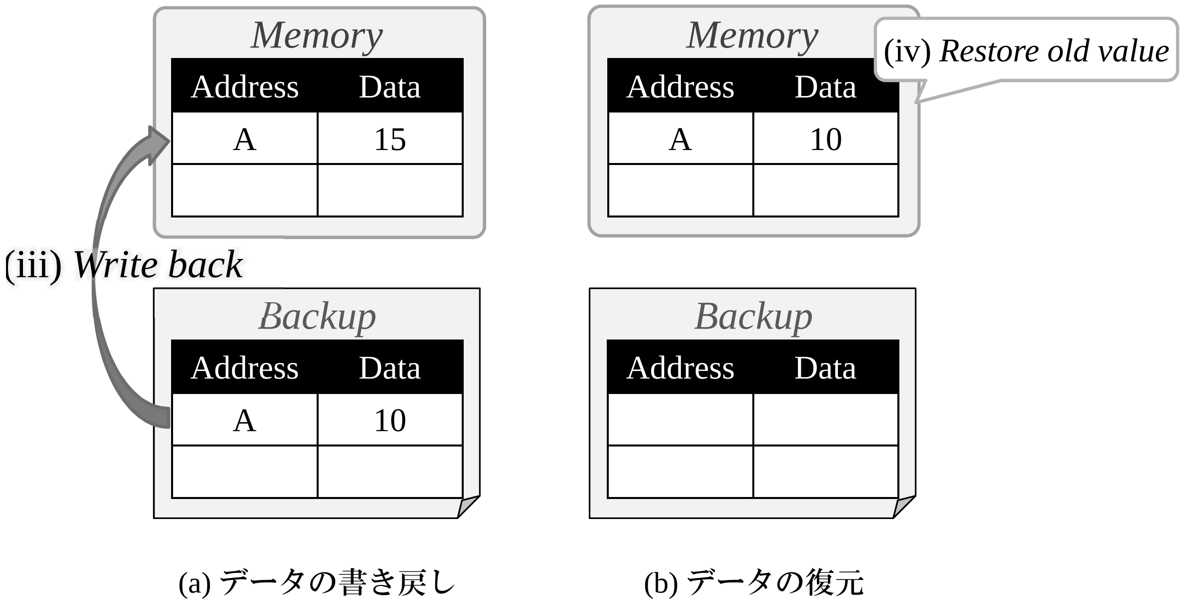


図 2: アボート時のバージョン管理

リ状態を復元することができる（iv）．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を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

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トランザクション毎に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** ビットおよび **write** 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キャッシュラインに対する Read アクセスおよび Write アクセスが発生した場合にそれぞれセットされ，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．このプロトコルでは，あるスレッドがメモリにアクセスする場合，アクセス対象となるラインが他スレッドによって既にアクセス済みであるかどうかをディレクトリに問い合わせる．既にアクセスされていた場合，拡張したプロトコルにおいて，各スレッドは要求を受信すると，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，他のトランザクションとの競合を監視する．このとき，以下の 3 パターンのアクセスが発生した場合に競合として判定される．

Read after Write（RaW）：

あるトランザクション内で Write アクセスが発生し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である．つまり，write ビットがセットされているアドレスに対する Read アクセス要求がプロトコルにより検出された場合である．この要求が許可されてしまうと，トランザクション内で更新した値がコミットされる前に他のトランザクションから読み出されるため，トランザクションの性質を満たさない．

Write after Read（WaR）：

あるトランザクション内で Read アクセスが発生し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である．つまり read ビットがセットされているアドレスに対する Write アクセス要求がプロトコルにより検出された場合である．この要求が許可されてしまうと，トランザクションがコミットされる前に，その内部で読み出した値が他のトランザクションによって変更されることで，それらのトランザクションを直列化して実行した結果と異なる可能

性があるため，トランザクションの性質を満たさない．

Write after Write（WaW）：
あるトランザクション内でWriteアクセスが発生し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である．つまりwriteビットがセットされているアドレスに対するWriteアクセス要求がプロトコルにより検出された場合である．この要求が許可されてしまうと，トランザクションがコミットされる前に，その内部で更新した値が他のトランザクションによって変更されるため，WaRと同様にトランザクションの性質を満たさない．

以上のような競合パターンが検出されると，要求を送信したスレッドに対して$NACK$が返信される．一方で競合が検出されなかった場合はACKが返信される．なお，実際に$NACK$およびACKを送信するのは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送信するとして説明する．

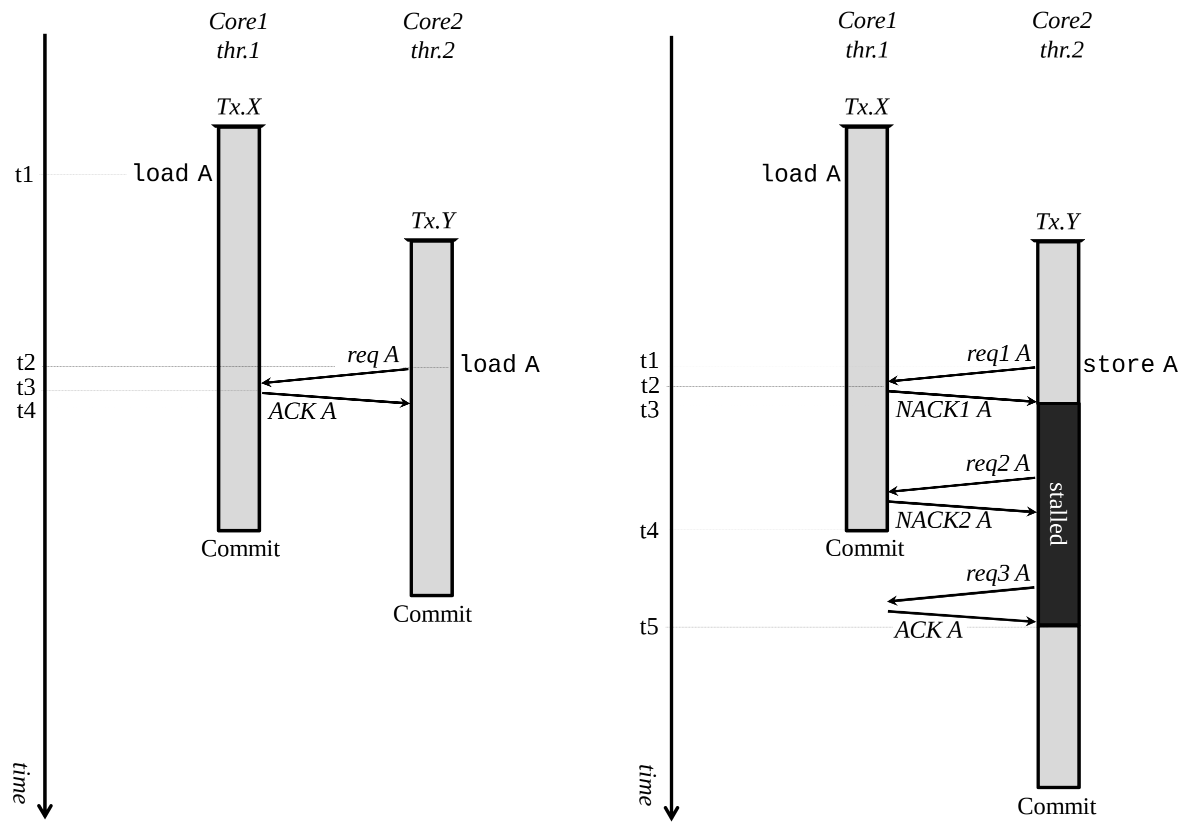
このようにして，$NACK$を受信したスレッドは自身のアクセスにより競合が発生したことを知ることができる．なお，この競合検出方式は，競合検査のタイミングによって以下の2つに大別される．

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存在しないか検査する．

Lazy Conflict Detection： トランザクションをコミットしようとした時点で，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発生していないか検査する．

前者のeager方式では，実際に競合が発生した際に即座にそれが検出されるのに対し，後者のlazy方式ではトランザクションのコミット時にしか競合の有無が検査されない．つまり，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り，無駄な実行が増加してしまう．したがって，eager方式の方が効率が良いと考えられるため，こちらの方式を本研究の対象とする．

Eager方式では競合が発生した場合，次の2つの方法のどちらかにより競合が解決される．1つ目は，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方法である．この方法では，アボートが頻繁に発生する場合があるため，性能が大きく低下してしまう可能性がある．これに対し2つ目は，複数の競合が発生した場合にフラグを用いてアボート対象のトランザクションを決定し，競合を解決する方法である．この方法では，即座にアボートする場合と比べてトランザクション内の命令がより多く並列実行されるため，一般的なHTMではこの方法が採用されている．なお，トランザ



(a) 競合なし　　(b) 競合あり

図 3: 競合の検出

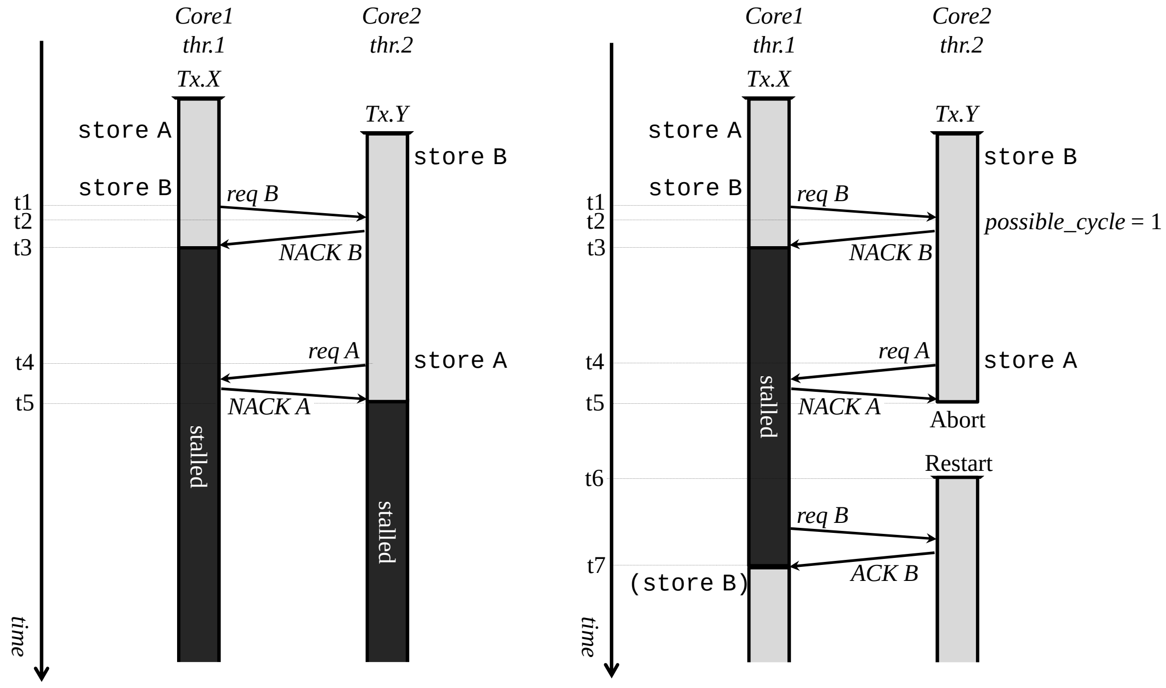
クションをストール中のスレッドは競合したアドレスに対するリクエストを競合相手に送信し続けることで，相手のトランザクションが終了したかどうかを監視する．一方で，競合が検出されなかった場合は従来の一貫性プロトコルに従う．たとえば，無効化リクエストに対しては ACK が返信され，共有リクエストに対しては共有されるデータが返信される．

ここで，**図 3** を用いて eager 方式の競合検出について具体的に説明する．図 3 中の縦軸は下に向かって時間が進むことを示している．また，Core1 と Core2 はそれぞれコアを，*thr.1* と *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トランザクション *Tx.X* と *Tx.Y* を投機的に実行しているとする．また，図 3 中でアクセス対象となっているアドレス `A` は共有メモリ上のアドレスであるとする．まず，競合が発生しない場合について図 3(a) を用いて説明する．図 3(a) では，まず *thr.1* が *thr.2* より先にアドレス `A` に対するロード命令を実行する，すなわち Read アクセスを

行う（時刻 t1）．この時点ではアドレス A は他スレッドからアクセスされていないため，*thr.1* は `load A` を実行することができる．この後 *thr.2* がアドレス A に対してロード命令を実行しようとする場合，*thr.2* は命令を実行する前にアドレス A にアクセスするためのリクエストを *thr.1* に送信する．*thr.1* はこのリクエストを受信すると，*thr.2* がアドレス A にアクセスしようとしていることを知る（t3）が，前述した 3 つの競合パターンに当てはまらないため競合は検出されない．このため，*thr.1* は *thr.2* に対して ACK を返信し，ACK を受信した *thr.2* は `load A` を実行することができる（t4）．

次に，図 3(b) を用いて競合が発生する場合について説明する．図 3(b) では，図 3(a) の場合と同様に *thr.1* がアドレス A に対するロード命令を実行する．その後，*thr.2* がアドレス A に対してストア命令を実行しようとする，すなわち Write アクセスを行おうとする（t1）．このとき，*thr.2* は命令を実行する前に *thr.1* へアドレス A にアクセスするためのリクエストを送信する．しかし，このアクセスは前述した競合を引き起こすアクセスパターンのひとつである WaR アクセスであるため，競合として検出される．このため，*thr.2* からのリクエストを受信した *thr.1* は，競合したことを通知するために *thr.2* へ $NACK$ を返信する（t2）．そして，この $NACK$ を受信した *thr.2* は自身が実行中の $Tx.Y$ をストールする（t3）．この後，*thr.2* は $Tx.Y$ をストールしている間もリクエストを再送し続け，アドレス A へのアクセス許可を待つ．続いて $Tx.X$ の実行が進み，*thr.1* がこの $Tx.X$ をコミットしたとする（t4）．この後， *thr.2* は再送したリクエストに対する ACK を *thr.1* から受信することで， アドレス A へアクセスできるようになったことを知る（t5）．これにより，*thr.2* は $Tx.Y$ をストール状態から復帰させ，実行を再開する．

しかし図 4(a) で示すように，2 つのスレッドがお互いの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．この例では，2 つのスレッド *thr.1* と *thr.2* がそれぞれトランザクション $Tx.X$ および $Tx.Y$ を投機的に実行している．まず，*thr.1* が $Tx.X$ の実行を開始した後に *thr.2* が $Tx.Y$ の実行を開始しており，*thr.1* が `store A` を実行した後に *thr.2* が `store B` を実行済みであるとする．ここで，*thr.1* が `store B` を実行しようとする場合，*thr.1* は *thr.2* に対してアドレス B に対するアクセスリクエストを送信する（t1）．このリクエストを受信した *thr.2* は競合を検知するため $NACK$ を *thr.1* に返信する（t2）．一方で $NACK$ を受信した *thr.1* は $Tx.Y$ が終了するまで $Tx.X$ をストールする（t3）．図中では省略しているが，*thr.1* はアクセスの許可を受けるまで *thr.2* に対して定期的にリクエストを送信している．この後，*thr.2* が `store A` の実行を試みる（t4）と，*thr.1* は *thr.2* との競合を検知し，$Tx.Y$ を



(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 4: HTM におけるデッドロック状態の解決

ストールさせる（t5）．このように，並列実行される 2 つのスレッドは，互いに相手の実行が終了するまで自身のトランザクションをストールすることになり，これ以上処理を進めることができなくなる．

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．具体的には，デッドロックを起こしうると考えられるトランザクションが競合相手のトランザクションよりも遅く開始されていた場合，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 possible_cycle と呼ばれるフラグを保持させることで実現されている．ここで，possible_cycle フラグによりデッドロックを回避する例を図 4(b) に示す．*thr.2* は *thr.1* に $NACK$ を返信する際，*thr.1* が自身より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（t2）．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，トランザクションをアボートする（t5）．このように，開始時刻の遅いトランザクションがアボートの対象として選択され，先に実行が開始されたトランザクションを優先的にコミットできるため，全てのトランザクションで

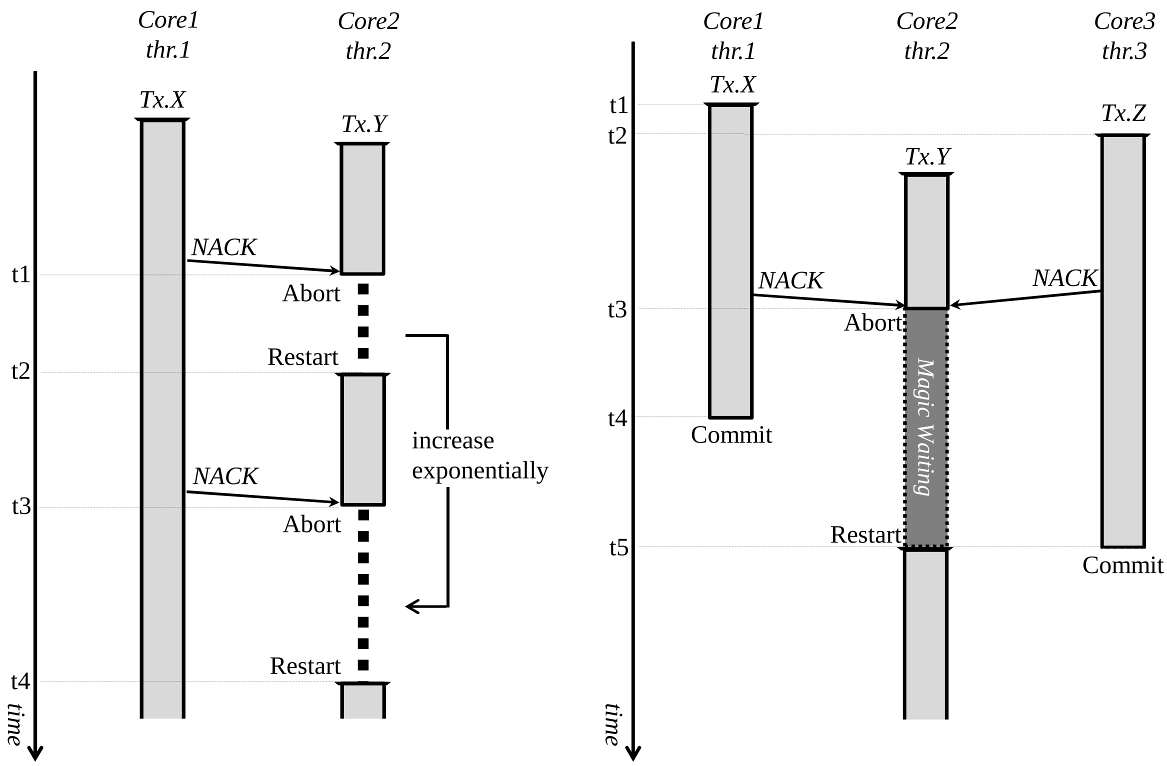
いずれ目的の共有リソースにアクセスできるようになる．その結果，トランザクションが飢餓状態に陥ることを防ぐことができる．続いて，$Tx.Y$ をアボートした *thr.2* はそのトランザクションを再実行する（t6）．また，*thr.2* がトランザクションをアボートしたことにより，*thr.1* はアドレス B にアクセスできるようになるため，$Tx.X$ のストール状態が解消される（t7）．

2.4 競合の再発抑制技術

前節で述べたアボート処理の後，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．したがって HTM には，競合の再発を抑制するための技術が複数存在する．その代表的なものが **exponential backoff** および **magic waiting** である．

まず exponential backoff は，トランザクションをアボートした際，次の再実行までの間に待機時間を設ける手法である．この待機時間は backoff と呼ばれ，アボートを繰り返す毎に指数関数的に増大させる．この exponential backoff の動作を図 **5**(a) を用いて説明する．図中では *thr.2* の実行する $Tx.Y$ がアボートされ，待機時間が設けられている（t1〜t2）．この待機処理が終わると $Tx.Y$ は再実行されるが，再びこの $Tx.Y$ のアボートが発生した場合，再び待機時間が設けられる（t3〜t4）．このとき，トランザクションのアボートが繰り返されてしまったため，前回よりも長い待機時間が設定される．このように，アボートが繰り返されるたびにより長い待機時間が設けられることで，競合の再発が抑制される．なお，待機時間が長くなり過ぎることを防ぐために，事前に規定した回数だけアボートを繰り返した場合，それ以降は待機時間が増大しないように設定される．また，トランザクションがコミットされる際，この待機時間はリセットされる．

一方で magic waiting は，トランザクションをアボートした場合，競合相手のトランザクションがコミットされるまで待機し続けることで，完全に再競合を防ぐ手法である．ここで，複数のスレッドと競合が発生した場合における magic waiting 手法の様子を図 5(b) を用いて説明する．図 5(b) では，まず *thr.1* が $Tx.X$ の実行を開始し（t1），次に *thr.3* が $Tx.Z$ の実行を開始している（t2）．これらの実行が進み，*thr.2* が *thr.1* および *thr.3* から $NACK$ を受信して $Tx.Y$ をアボートした場合（t3），受信した $NACK$ から競合相手のトランザクション開始時刻を取得し，その時刻が一番遅いスレッドがトランザクションをコミットするまで待機する．この例では，$Tx.X$ よりも $Tx.Z$ の開始時刻の方が遅いため，*thr.2* は $Tx.Z$ のコミットまでトランザクションを実行せずに



(a) Exponential backoff (b) Magic waiting

図 5: 競合の再発抑制技術

待機し続けなければならない．したがって，競合相手の 1 つである $Tx.X$ がコミットされた場合（t4）も待機し続ける．そして $Tx.Z$ がコミットされると，$thr.2$ は $Tx.Y$ を再実行することができる（t5）．このようにして，magic waiting を用いた場合は競合の再発を防止することができる．なお，この手法を用いた際には，並列実行可能なトランザクションも待機させてしまう場合があるため，並列度が低下し結果として性能に悪影響を及ぼしてしまう可能性がある．そのため，一般的な HTM では exponential backoff が用いられている．

しかしながら，この exponential backoff を用いた場合でも，性能に悪影響を及ぼす競合パターンが発生する可能性がある．本論文では，starving writer の発生と futile stall の発生という 2 つの競合パターンに注目し，その解決手法と実装モデルを 3 章および 4 章でそれぞれ述べる．

3 Starving Writer 解消手法

本章では，本論文で解決する1つ目の問題である**starving writer**[8]と呼ばれるトランザクションを発生させる競合パターンについて述べ，これを解消する手法の提案と実装方法について説明する．

3.1 Starving writer の発生

HTM では，ある特定の競合パターンが発生すると著しく性能が悪化する場合がある．その競合パターンの1つに大きく関わっているのがstarving writerと呼ばれるトランザクションの発生である．これは，あるアドレスに対して書き込もうとするwriterが，そのアドレスから値を読み出す複数のreaderにより妨げられ続けることにより発生する．

いま図**6**のように，3つのスレッド（*thr.1*〜*3*）が，それぞれトランザクションを実行する例を用いてstarving writerの発生する状況を説明する．なお，*thr.1* および *thr.3* は同一のトランザクション *Tx.X* を実行し，*thr.2* は *Tx.X* で Read アクセスされるアドレス `A` に対する書き込みを含む *Tx.Y* を実行するとする．まず *thr.1* が `load A` を実行済みの状態で，*thr.2* が `store A` を実行しようとしたとする．このアクセスは競合となるため，*thr.1* から $NACK$ を受信した *thr.2* は *Tx.Y* をストールする（t1）．この場合，*thr.1* が *Tx.X* をコミットもしくはアボートしない限り *thr.2* は `store A` を実行できない．次に，*thr.3* が `load A` を実行しようとした場合（t2），これは2.3節で述べたいずれのアクセスパターンにも該当しないため，競合は検出されない．これにより，*thr.1* および *thr.3* が実行中の2つの *Tx.X* が，ともにアボートもしくはコミットしない限り *thr.2* は再開することができない．この後，*thr.1* が *thr.2* と競合して *thr.1* の *Tx.X* がアボートされた場合（t3）でも，*thr.3* がすでにアドレス `A` にアクセスしているため，*thr.2* は実行を再開できない．そして *thr.1* は *Tx.X* の再実行後，再度 `A` にアクセスしてしまう（t4）．そのため，*thr.3* の *Tx.X* がアボートされた場合も *thr.2* は実行を再開できない．このように，同一アドレスの値を読み出すスレッドが複数存在することで，当該アドレスに対して書き込もうとしているスレッドが飢餓状態となる．この飢餓状態となったスレッドが実行するトランザクションをstarving writerと呼ぶ．また，このstarving writerの競合相手である *Tx.X* はアボートを繰り返し（t5，t6），この間starving writerは飢餓状態であり続けてしまうため，大幅な性能低下の原因となる．

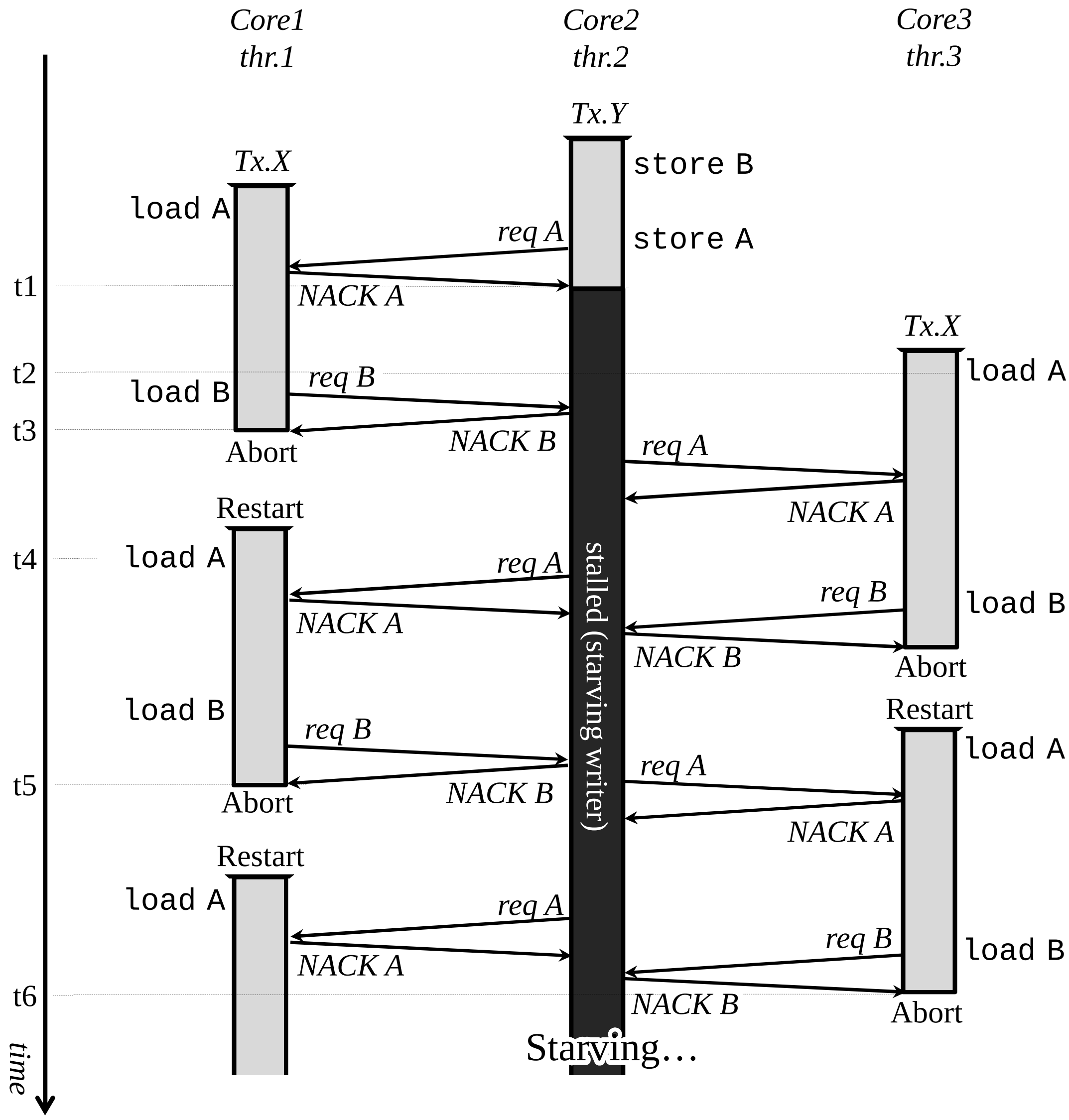


図 6: Starving writer の発生

なお，並列実行されるトランザクションの増大にともない，読み出しを行うスレッドの数も増えると考えられる．その際，それらすべてのスレッドがトランザクションをアボートあるいはコミットしてリソースを解放しない限り，書き込もうとしているスレッドは実行を再開することができない．したがってこの starving writer の発生により性能が大きく低下してしまうと考えられる．

この競合パターンは 2.4 節で述べた exponential backoff または magic waiting によって対処可能である．しかし，HTM で一般的に用いられている exponential backoff で

はトランザクションのアボート後，その回数が少ないうちは待機時間が短く設定されるため，starving writer が解決されるまでに reader が何度もアボートを繰り返してしまう．これを避けるために backoff 待機時間の増大率を大きくすることも考えられるが，starving writer に関与している reader 数に応じた増大率を設定しなければ，backoff の期間が長くなり過ぎてしまい性能が著しく低下するおそれがある．しかし，これら reader はアボートなどにより常にその数が変動するため，reader 数を把握することも一般に容易ではない．一方 magic waiting では，アボートを繰り返していない場合に並列実行可能なトランザクションを待機させてしまう可能性があるため並列度が低下してしまう．

3.2 Starving writer の解消

本論文では，前節で述べた starving writer の発生を抑制するための手法を提案する．この手法では，reader が writer を競合相手としてある条件を満たし，アボートを繰り返す場合，その reader の実行に magic waiting を適用することで相手 writer を優先的にコミットさせる．これにより，writer の飢餓状態および reader に発生するアボートが抑制できると考えられる．

さて，starving writer は WaR 競合パターンが存在する場合に発生する．また，2.3 節で述べたように，アボート処理までに少なくとも 2 つのキャッシュラインで競合が発生する．これを踏まえ本節では，starving writer が発生したと考えられる場合に writer の実行を優先させる 3 つのモデルを提案する．なお各提案モデルでは，それぞれ 2 つの条件が設けられており，これらの条件を満たした reader に magic waiting を適用することで相手 writer の実行を優先させる．

モデル 1：同一 writer との競合アドレスの組が一致

あるトランザクションが以下に示す 2 つの条件をともに満たす場合，競合相手を starving writer であると判定し，reader に magic waiting を適用することで競合相手の writer を優先的にコミットさせる．

- 条件 S-I：自身が Read アクセス済みのアドレスに対して他スレッドが Write アクセス要求を発行することにより WaR 競合が発生
- 条件 S-II：同一の writer との間の競合によって発生した，直近の過去 2 回のアボートに関与したアドレスの組 {競合アドレス 1，競合アドレス 2} が一致

なお 2.3 節で説明したように，アボートが引き起こされるまでに，possible_cycle フラグをセットする原因になった競合と，アボート発生の直接的な原因となった競合の 2

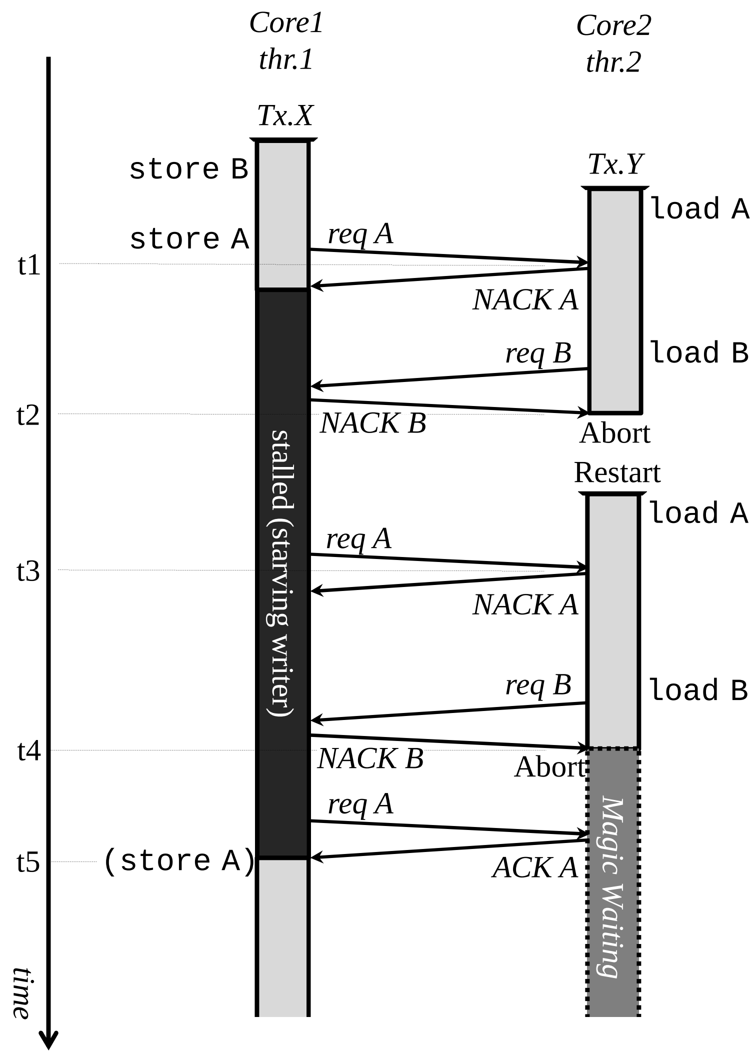


図 7: Starving writer 発生時の動作モデル

つのアドレス競合が発生するが，条件 S-II の「アボートに関与したアドレスの組」とは，これら 2 つの競合それぞれにおける対象アドレスの組を指す．

これら 2 つの条件について**図 7** の starving writer が発生する場合の例を用いて説明する．例ではまず *thr.2* が `load B` を実行し，次に *thr.1* が `store B` を実行しようとして競合が発生する（t1）．これは WaR 競合であるため，条件 S-I を満たす．その後 *thr.2* が実行する *Tx.Y* は，*thr.1* によって 2 度アボート（t2，t4）させられており，アボートに関与したアドレスの組合せがともに {`A`, `B`} となっている．したがって条件 S-II を満たす．このように両方の条件を満たした場合，Write アクセスをしようとしていたトランザクション *Tx.X* を starving writer であると見なし，Read アクセスしていたス

レッドに対して magic waiting を有効にすることで，飢餓状態となっていたトランザクションを優先的に実行させる（t5）．

モデル2：同一 writer との競合アドレスが一致

提案モデル1ではアボートに関与したアドレスの組が連続で同じかどうかを条件 S-II として用いた．しかし，アボートが繰り返される場合，possible_cycle フラグをセットする原因となる競合アドレスが前回とは異なる場合がありうる．

そのような状況が発生する場合について**図8**を用いて説明する．図8ではまず，*thr.2* が `store A` を実行しようとして *thr.3* と競合を起こし，*Tx.Y* をストールする (t1)．続いて，*thr.3* が `load B` を実行しようとして（t2）*thr.2* と再び競合を起こし，*thr.3* は *Tx.X* をアボートする．このアボートでは，possible_cycle flag がセットされる原因となったアドレスは `A` であり，アボートに直接関与したアドレスは `B` である．

次に，*thr.1* の実行する *Tx.X* と *thr.2* の実行する *Tx.Y* で競合が発生した場合，*thr.2* は *Tx.Y* をアボートする（t3）．この *Tx.Y* のアボート処理中に *thr.1* が *Tx.X* をコミットした場合（t4），メモリに反映された結果が *Tx.Y* の実行に影響し，*Tx.Y* の制御フローが変化したとする．この後，制御フローの変化した *Tx.Y* を *thr.2* が再実行（t5）した後，`store C` を実行しようとして（t6）競合が発生する．そして，*thr.3* は `load B` を実行しようとして（t7）*thr.2* と競合し，*Tx.X* をアボートする（t8）．前回のアボートでは，possible_cycle flag がセットされる原因となったアドレスは `A` であったが，このアボートではフローの変化により，前回とは異なるアドレス `C` がその原因となっている．一方，アボートに直接関与したアドレスはともに `B` である．

この例のように，フラグをセットする原因となった競合アドレスのみが前回と異なる場合があるが，この場合には提案モデル1では対応することができない．したがって，提案モデル1の条件 S-II を以下のように緩和することで，図8のような場合も解決する2つめのモデルを考える．

- 条件 S-II′：同一の相手による直近の過去2回のアボートにおいて，そのアボートに直接関与するアクセス対象アドレスが同一

すなわち，possible_cycle フラグをセットする原因となったアクセス対象アドレスの一致を必要としないよう，条件 S-II を変更する．これと，提案モデル1の条件 S-I を併用することで，starving writer を解消する．図8の例では，*thr.3* の実行する *Tx.X* のアボートはともにアドレス `B` へのアクセス（t2，t7）を直接的な原因として発生しているため条件 S-II′ に該当し，*thr.3* に magic waiting が適用される．

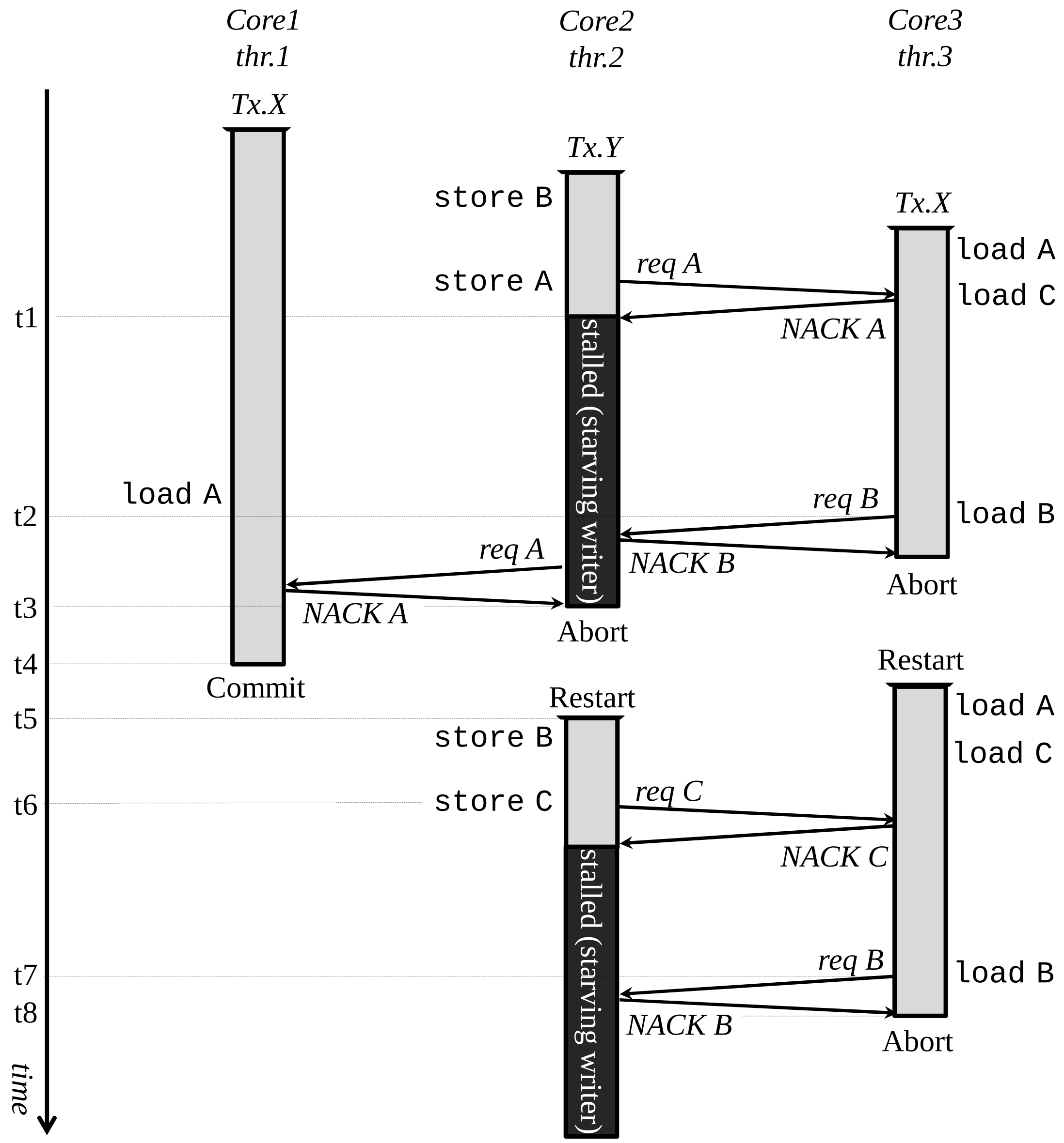


図 8: 制御フローが変化する場合

モデル 3：任意の writer との競合アドレスが一致

提案モデル 1 および 2 では，reader が同一の writer と競合してアボートを繰り返したかどうかを条件に用いた．しかし，この競合相手が同一であるかどうかを条件に用いなくても良い場合がある．たとえば，図 6 のように 1 つの starving writer が存在する場合，各 reader は同一相手との競合によりアボートを繰り返すため，各 reader は発生したアボートがどのトランザクションとの競合によるものなのかを考慮しなくてよ

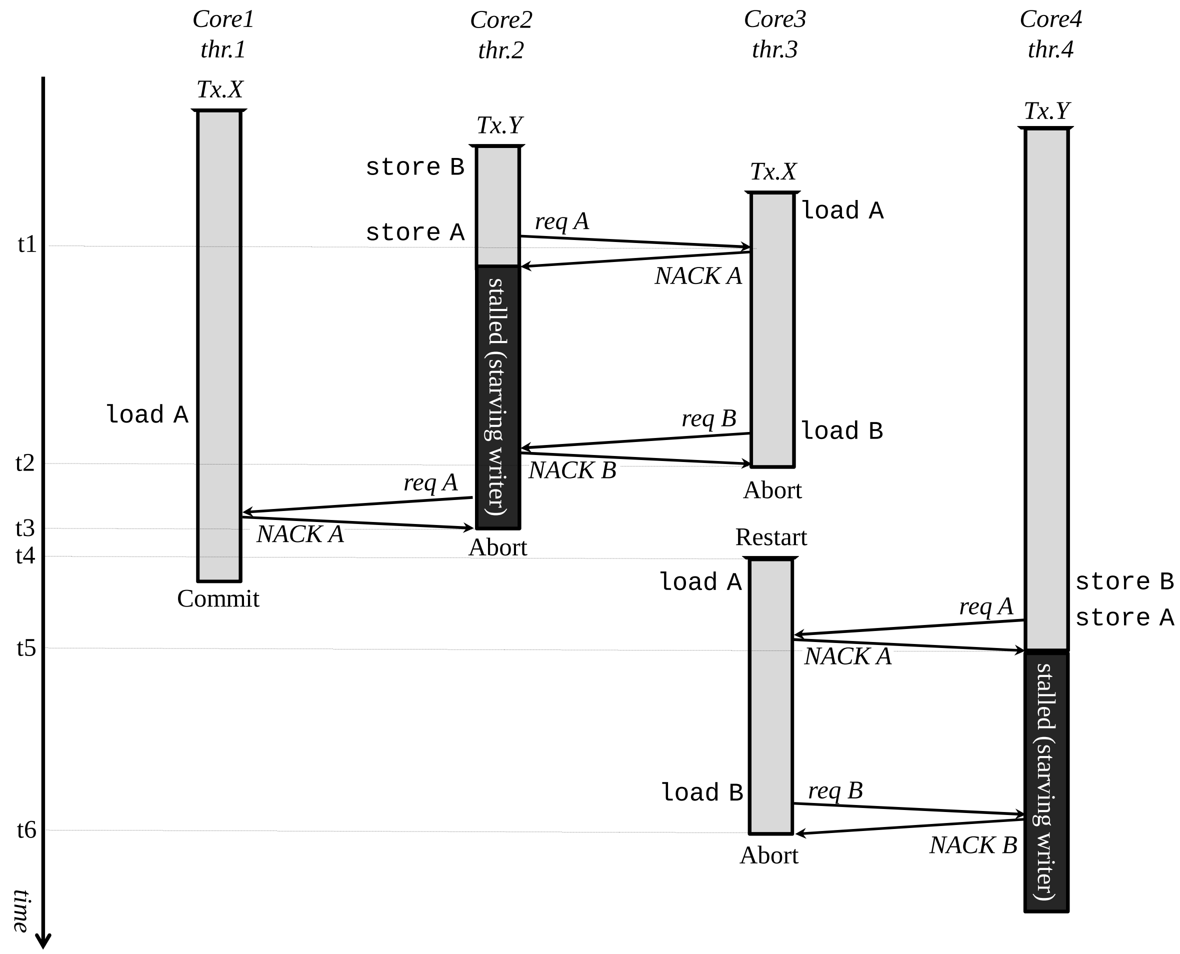


図 9: starving writer が複数存在する場合

い．また，starving writer となりうるトランザクションが複数存在する場合も競合相手が同一であるかどうか考慮しなくても良い場合があると考えられる．その理由を図 **9** を用いて説明する．

この例では，*thr.1* および *thr.3* が *Tx.X* を実行し，*thr.2* および *thr.4* が *Tx.Y* を実行している．まず，*thr.3* が `load A` を実行した後，*thr.2* が `store A` を実行しようとして *thr.3* と競合したとする（t1）．次に，*thr.3* が `load B` を実行しようとして *thr.2* と競合した場合，*thr.3* は *Tx.X* をアボートする（t2）．この後，*thr.2* の実行する *Tx.Y* が *thr.1* との競合によりアボートされるとする（t3）．そしてこのアボートされた *Tx.Y* が再実行されるまでの間に，*thr.3* が *Tx.X* を再実行し（t4），*thr.4* と 2 回の競合を起こしたとする（t5，t6）．すると，*thr.3* は先のアボート時とは異なる相手との競合により *Tx.X* を再びアボートしてしまうが，アボートの直接の原因となったアドレスは

前回のアボート時と変わらず`A`である．このとき提案モデル 2 の場合では，*thr.3* の競合相手が変わったことにより，条件 S-II′ が満たされない．そのため，*thr.3* は magic waiting を適用せずに *Tx.X* を再実行してしまい，*thr.2* および *thr.4* と再び競合を起こすと考えられる．

そこで，このような場合にも reader に magic waiting を適用することで starving writer を解消するために，提案モデル 2 に対して競合相手を考慮しないように条件を簡略化したモデルを考える．これを実現するため，提案モデル 2 の条件 S-II′ において，競合相手に関する条件を次のように緩和する．

- 条件 S-II″：競合相手を問わず，直近の過去 2 回のアボートにおいて，そのアボートに直接関与するアクセス対象アドレスが同一

なお提案モデル 2 と同様に，提案モデル 1 の条件 S-I を併用する．

3.3 追加ハードウェアと動作モデル

前節で述べた提案モデルを実現するために，既存の HTM を拡張し，以下に示す 3 つの機構を各コアに追加する．また，追加した機構を**図 10** に示す．

WaR flags： WaR 競合パターンの発生の有無を記憶する．自スレッドがすでにロードを実行済みであるアドレスに対し，他スレッドがストアを実行しようとした際の競合発生時にセットされる．相手スレッドごとに WaR 競合が存在するかを記憶するため，総スレッド数 n に対して n bit 必要であり，アボートおよびコミット時にクリアされる．

Conflict Table（C-Tbl）： スレッド番号をインデクスとし，そのインデクスに対応するスレッドとの間に起こった直近の競合における対象アドレスを記憶する．競合発生時に参照され，現競合の対象アドレスと比較される．アドレスが一致した場合は後述の M-W flags の状態を更新し，一致しない場合は現競合アドレスでエントリを上書きする．提案モデル 1 は 2 つのアドレスを条件判定に利用するため，深さ n のこのテーブルを各コアに 2 つ（C-Tbl1，C-Tbl2）保持させる．そして，競合したアドレスに対して先にアクセスしたのが自スレッドである場合は C-Tbl1，競合相手スレッドである場合は C-Tbl2 を用いる．一方，提案モデル 2 ではアボートの直接の原因となったアドレスのみを保持するため C-Tbl は 1 つとなり，提案モデル 3 では相手スレッドを区別しないため C-Tbl は 1 つかつ深さ 1 となる．なお，テーブル内の情報はコミット時のみクリアされる．

Magic Waiting flags（M-W flags）： Magic waiting を有効にするかどうかを示す

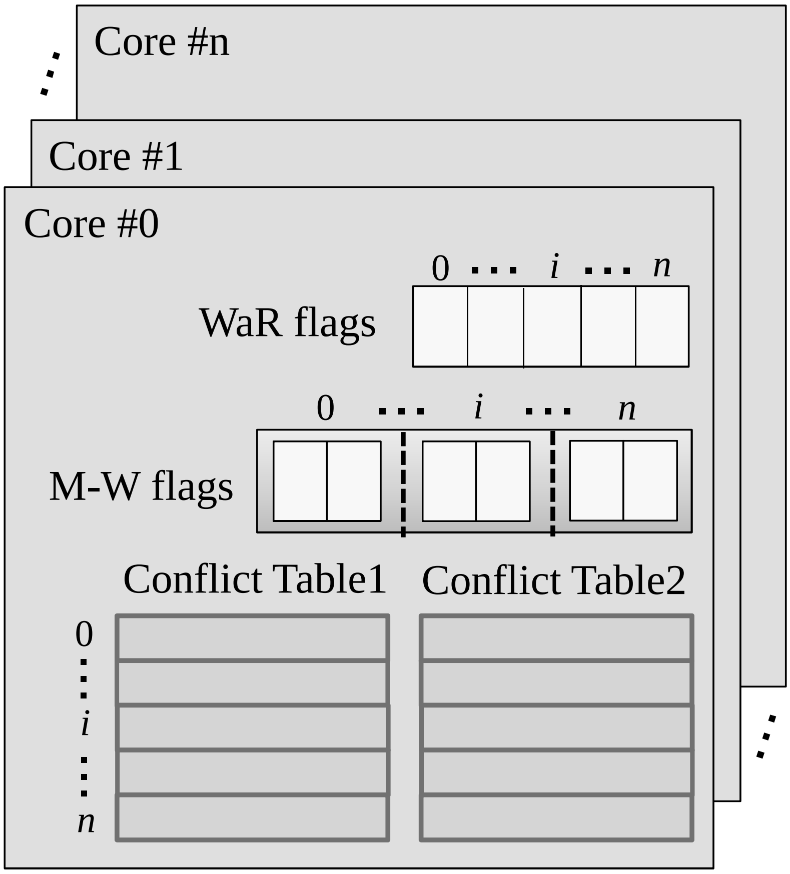


図 10: 追加機構

2n bit からなるビット列で，各スレッドに対して 2 ビットずつ使用する．C-Tbl1 で比較したアドレスが一致した場合は 1 ビット目，C-Tbl2 では 2 ビット目のビットがセットされ，アボート時にこれら 2 つのビットが両方セットされている場合，magic waiting を有効にする．コミットおよびアボート時にクリアされる．なお提案モデル 2 および 3 では，登録済みのアドレスと競合したアドレスが一致した場合に magic waiting を有効にするかどうか判定すれば良いため，M-W flags を必要としない．

ここで，**図 11** の starving writer が発生する場合の例を用いて，*thr.2* における提案モデル 1 のハードウェアの動作を説明する．まず，*thr.2* が `load A` を実行し，その後に *thr.1* が `store A` を実行しようとした場合，WaR パターンの競合が検出される（t1）．したがって *thr.2* では，競合相手のスレッド *thr.1* に対応する WaR flags がセットされ，スレッド番号 1 をインデクスとして C-Tbl が参照される．なお，今回競合が発生したアドレス `A` に先にアクセスしたのは *thr.2* であるため，C-Tbl1 が参照される．このと

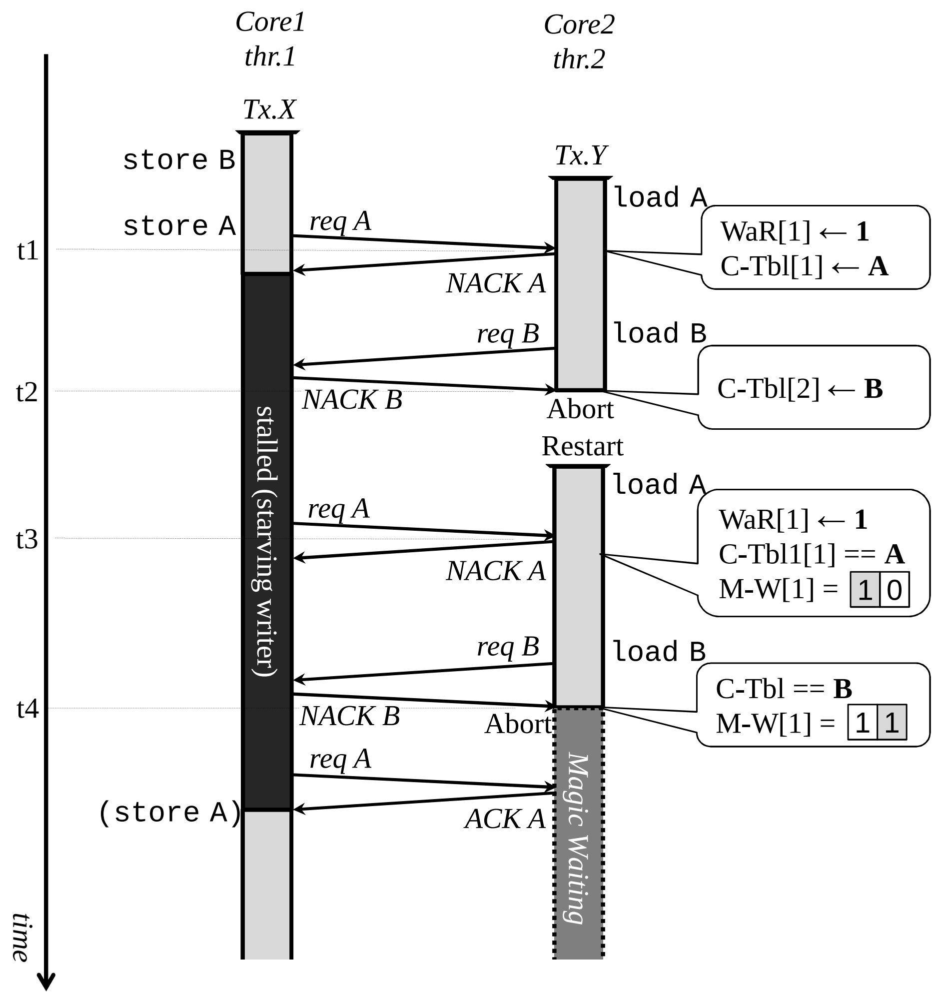


図 11: 追加したハードウェアの状態変移

き，C-Tbl1[1] にはアドレスが未登録であるため，アドレス A が登録される．

次に，*thr.2* が `load B` を実行しようとして競合が発生した場合，アドレス B へ先にアクセスしたのは *thr.1* であるため，先ほどとは別のテーブルである C-Tbl2 が参照される．このとき *thr.1* に対応するアドレスは C-Tbl2 にはまだ登録されていないため，アドレス B が登録される（t2）．そして *thr.2* はデッドロックを回避するために，*Tx.Y* をアボートする．なお，アボート後はすべての競合が解決されるため，WaR flags はクリアされる．

続いて *thr.2* が *Tx.Y* を再実行し，アドレス A に対するアクセスで競合すると（t3），時刻 t1 のときと同様に WaR flags がセットされ，アドレス A が C-Tbl1 に登録されているか参照される．今回はすでに同一のアドレスが登録されているため，M-W flags

がセットされる．なお，C-Tbl1 に登録されているアドレスと現競合の対象アドレスが一致するため，M-W flags は左ビットがセットされ，`10` となる．次に *thr.2* が `load B` を実行し，競合が発生すると（t4），時刻 t3 のときと同様に C-Tbl2 が参照される．そして，登録済みのアドレスと今回競合したアドレス `B` とが一致するため，今度は M-W flags の右ビットがセットされる．この結果 M-W flags は `11` と両ビットがセットされている状態になるため，magic waiting が有効となる．

一方提案モデル 2 では，提案モデル 1 と比べ C-Tbl が 1 つ少なく済み，C-Tbl1 へのアドレス `A` の登録および参照の処理をする必要がなくなる．また，提案モデル 3 は提案モデル 2 と比べて C-Tbl の構造が異なるが，2 者間によるアボートの繰返し時には提案モデル 2 と同じ動作となる．なお，これら 2 つのモデルでは M-W flags が不要となるため，M-W flags に対する処理も行う必要がなくなる．

4　Futile Stall 防止手法

本章では，本論文で解決する 2 つ目の問題である futile stall の発生について述べ，これを防止する手法の提案と実装方法について述べる．

4.1　Futile Stall の発生

競合を頻繁に引き起こす複数のトランザクションが並列に実行されるとき，開始時刻の遅いトランザクションによって開始時刻の早いトランザクションがストールさせられる状況が頻繁に発生する．この競合の後，これらのトランザクション間で再度競合が発生した場合，開始時刻の遅いトランザクションがアボートされる．この場合，結果的には開始時刻の早いトランザクションをストールさせる必要がなかったことになる．このようなストールを futile stall という．

ここで**図 12** のように，3 つのスレッド（*thr.1*〜*3*）が，それぞれ同じトランザクション *Tx.X* を実行する例を用いて futile stall を説明する．なお，futile stall はロード命令とストア命令に依存せず発生するため，命令の表記を省略する．まず，*thr.1* によって既にアクセスされたアドレスに *thr.2* がアクセスしようとした場合，*thr.2* が実行する *Tx.X* はストールする（t1）．次に，*thr.2* が既にアクセス済みのアドレスに *thr.1* がアクセスしようとした場合，*thr.1* の実行する *Tx.X* の方がその開始時刻が遅いため，このトランザクションがアボートされる（t2）．このとき，*thr.1* が実行していたトランザクションのアボートにより資源が開放されたため，*thr.2* は *Tx.X* の実行を再開する．しかし，*thr.2* はトランザクションをストールさせている間，*thr.1* の実行するト

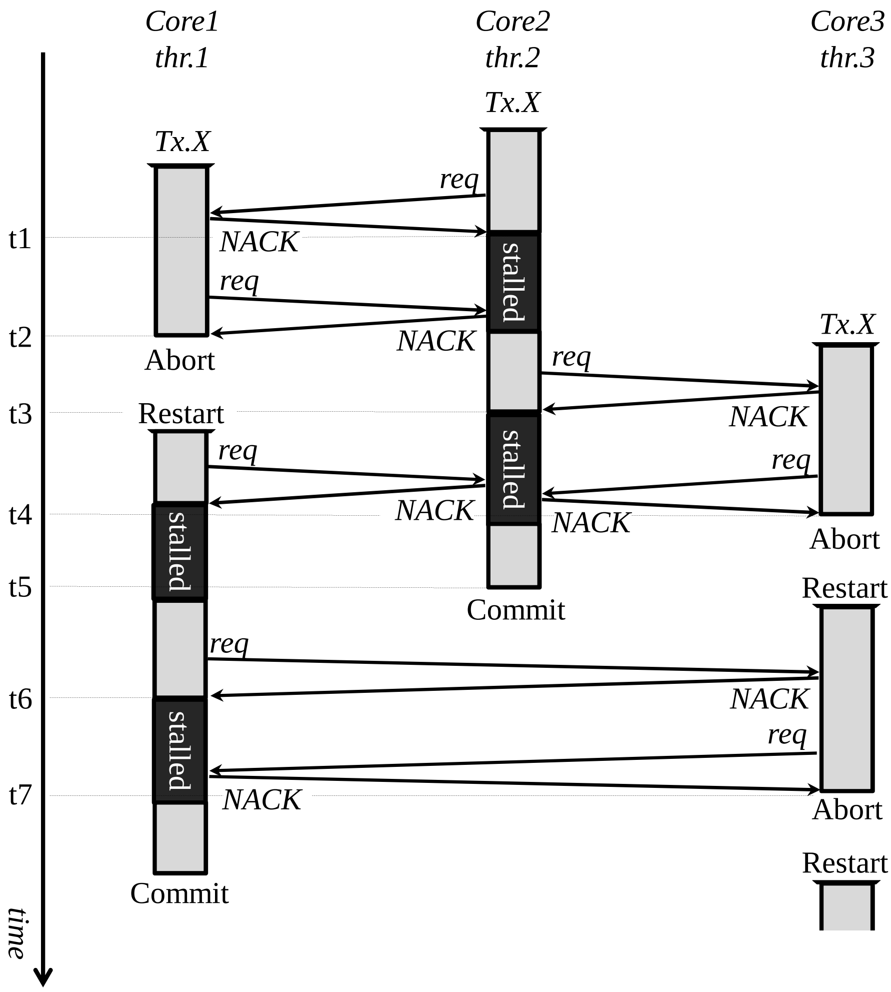


図 12: Futile Stall の発生

ランザクションがコミットされるのを待っていたにも関わらず，このトランザクションはアボートしてしまったため，結果的には *thr.2* の実行するトランザクションをストールさせる必要がなかったこととなる．このようなストールを futile stall という．

この後，*thr.2* が *thr.3* と競合（t3 および t4）した場合，さきほどと同様に *thr.2* の実行する開始時刻の早い *Tx.X* のストールは futile stall となり，その競合相手である *thr.3* の実行する *Tx.X* はアボートされる（t4）．続いて，*thr.2* が *Tx.X* をコミットした場合（t5），競合相手のトランザクションである *thr.1* および *thr.3* の *Tx.X* 同士で競合が発生し，*thr.1* の *Tx.X* はストールし（t6），*thr.3* の *Tx.X* はアボートされてしまう（t7）．

このように，競合を頻繁に引き起こす可能性のある複数のトランザクションが並列

に実行される場合は futile stall が多く発生する．また，*thr.3* の実行するトランザクションのように，アボートを繰り返してしまうトランザクションも発生する．なお，並列実行されるトランザクションの増大にともない，競合が発生する可能性がさらに増える．その結果としてアボートや futile stall の発生も増えるため，性能が大きく低下すると考えられる．

4.2　Futile Stall の防止

前節で述べた futile stall が多く発生する場合，並列実行されるトランザクション間で競合が頻繁に発生していると考えられる．その場合，図 12 の *thr.3* の実行するトランザクションのようにアボートの繰り返しが発生する．そこで本節では，アボートを一定回数繰り返したトランザクションを競合が頻発するトランザクションと判断し，そのトランザクションを逐次的に実行する手法を提案する．

ここで，図 12 で競合を頻発させた *Tx.X* が，逐次的に実行される場合の例を**図 13** に示す．例ではまず，*thr.2* が *Tx.X* の実行を開始し，次に *thr.1* が *Tx.X* の実行を開始しようとする（t1）．*thr.1* が実行しようとした *Tx.X* が逐次的な実行の対象となっている場合，*thr.1* は *Tx.X* の実行を開始せずに待機状態になる．続いて *thr.3* が *Tx.X* の実行を開始しようとした場合も同様に，*thr.3* も *Tx.X* の実行を開始せずに待機状態となる（t2）．その後，*thr.2* が *Tx.X* をコミットした場合，*thr.1* は *Tx.X* の実行を開始する（t3）．*thr.1* が *Tx.X* をコミットする場合も同様に *thr.3* が *Tx.X* の実行を開始し（t4），*thr.3* はコミットまで実行を進めることができる（t5）．

このように，競合を頻発させるトランザクションを逐次的に実行することで，本来そのトランザクションの実行によって発生するアボートの頻発を防ぐことができ，futile stall も抑制できると考えられる．しかし，実行命令数の多いトランザクションを逐次的に実行した場合には，逐次実行の時間が長くなり，性能が悪化する可能性がある．したがって，ある程度実行命令数の少ないトランザクションのみを逐次実行の対象とするよう考慮しなければならない．これらを踏まえ，あるトランザクションが以下に示す 2 つの条件を満たした場合，そのトランザクションを逐次実行の対象とする．なお，これらの条件で示されている *A-tx* および *L-inst* はそれぞれトランザクションのアボート回数およびトランザクションの実行命令数の閾値を表している．

- 条件 F-I：トランザクションをコミットするまでに，閾値 *A-tx* 以上アボートを繰り返す
- 条件 F-II：トランザクションをコミットするとき，そのトランザクション内で実

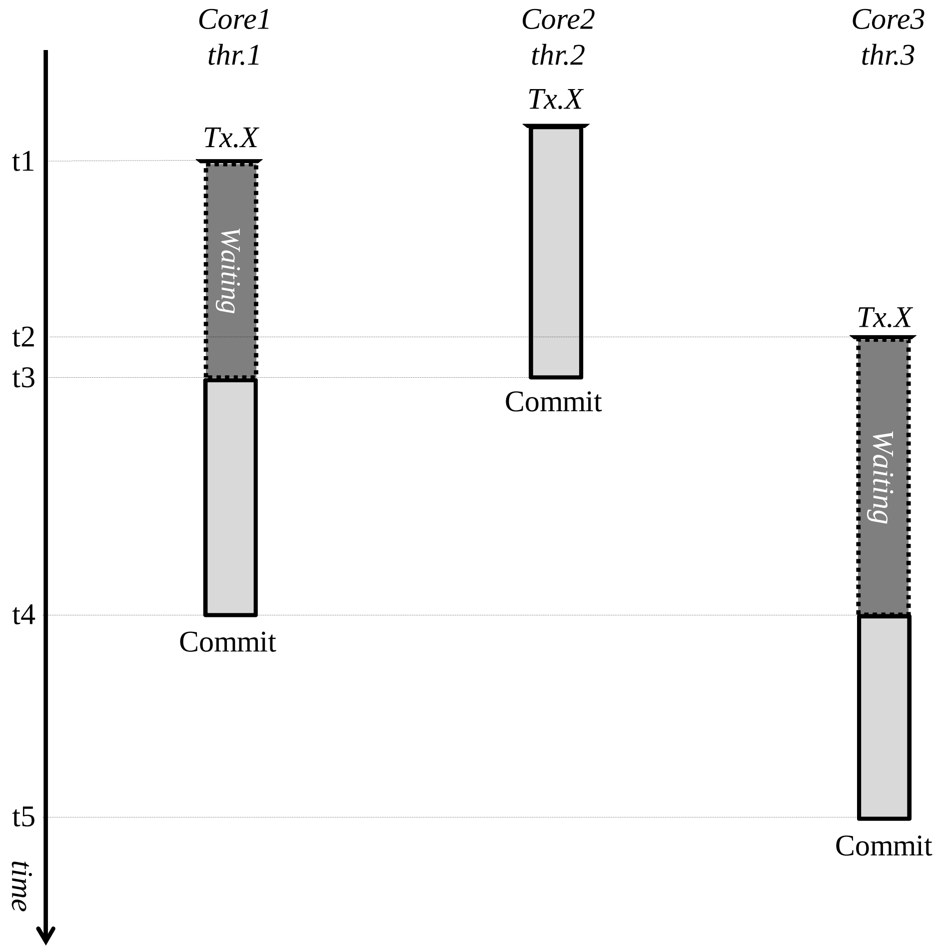


図 13: Futile stall 防止手法の実行モデル

行された命令数が閾値 *L-inst* 以下である

これらの条件を満たすトランザクションが複数存在する場合，それらのトランザクションを逐次的に実行する．ただし，トランザクションの実行中に OS による割り込み処理が発生した場合，トランザクションの開始からコミットまでに実行された命令の数はトランザクション内の実行命令数より多くなってしまう．そのため，あるトランザクションの開始からコミットまでに実行された命令数が閾値 *L-inst* より多い場合があったとしても，そのトランザクションを逐次的に実行することで性能向上が得られる可能性がある．

そこで，割り込み処理が原因で条件 F-II を満たさなかったトランザクションを救済するために，閾値 *L-inst* より値の小さい閾値 *S-inst* を設ける．たとえば，あるトランザクションが実行され，そのトランザクションは条件 F-II を満たさなかったとする．

その後，同じトランザクションが再び実行され，その実行命令数が今度は $S\text{-}inst$ よりも少なかった場合，そのトランザクションの実行命令数が多いと判定されたのはOSによる割り込み処理が原因であると判断し，そのトランザクションは条件F-IIを満たしたと判定する．

4.3 追加ハードウェアと動作モデル

前節で述べた動作を実現するために，まず逐次実行の対象トランザクションを決める必要がある．そして，その対象となったトランザクションの実行順序を制御しなければならない．本節では，これらの処理を実現するために追加したハードウェアおよびその動作モデルについて述べる．

4.3.1 逐次実行対象トランザクションの決定

逐次実行の対象トランザクションを決めるために，以下に示す5つの機構を各コアに追加する．また，追加した機構を**図14**に示す．なお，HTMでは，トランザクション毎にIDが割り振られており，このIDは個々のトランザクションを識別するために用いられる．下記に示すR-flags，Stx-flagsおよびLtx-flagsはそのIDをインデクスとしてトランザクション毎に管理される．

Abort Counter（A-Counter）: トランザクションの実行を開始してからコミットするまでの間に発生したアボートの回数をカウントする．このカウントされた値は，トランザクションのコミット時にリセットされる．

Recurrence flags （R-flags）: トランザクションのIDをインデクスとし，そのインデクスに対応するトランザクションがアボートを繰り返したかどうかを記憶する．前述したA-Counterの値が閾値 $A\text{-}tx$ 以上になった場合，実行中のトランザクションのIDに対応するビットがセットされる．なお，コミット後に再び同一のトランザクションが実行される可能性があるため，一度セットされたフラグはプログラムが終了するまでクリアされない．なお，図14で示したR-flagsは各コアに t bit用意された場合であり，トランザクションの数が t 個まで対応することができる．

Instruction Counter（I-Counter）: トランザクション内で実行された命令数をカウントする．制御フローの変化により，同一のトランザクションでも実行のたびにその命令数が変化する可能性がある．したがって，カウントされた値はアボートおよびコミット時にリセットされ，トランザクションの実行毎に命令数がカウントされる．

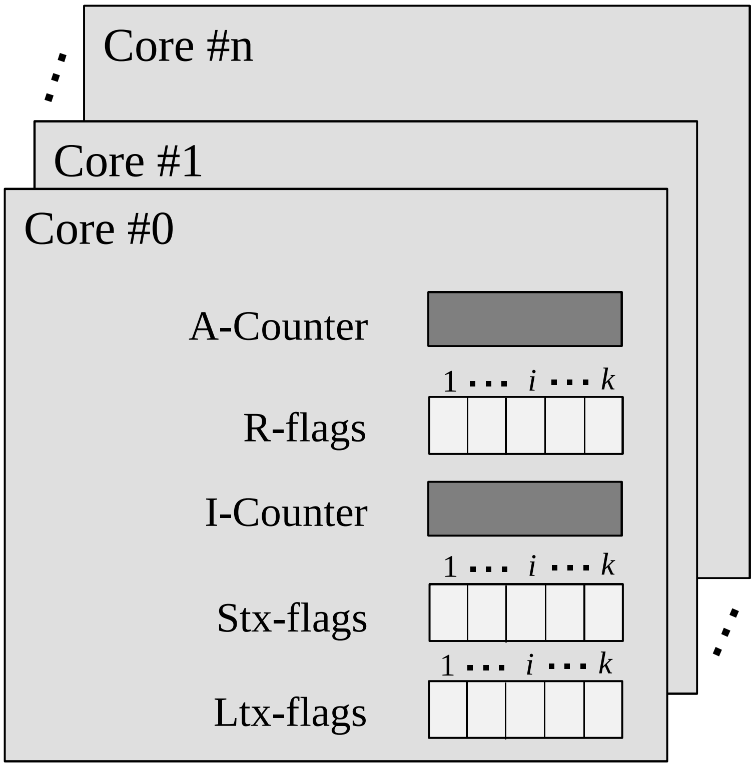


図 14: 逐次実行対象を決定する機構

Short Tx flags（Stx-flags）: トランザクションの ID をインデクスとし，そのインデクスに対応するトランザクションの実行命令数が少ないかどうかを記憶する．前述した I-Counter の値が，閾値 *L-inst* 以下の状態でコミットが発生した場合，実行中のトランザクションの ID に対応するビットがセットされる．これらのフラグはトランザクションを実行する毎に参照され，プログラムの終了までクリアされない．

Long Tx flags（Ltx-flags）: トランザクションの ID をインデクスとし，そのインデクスに対応するトランザクションの実行命令数が多いかどうかを記憶する．前述した I-Counter の値が，閾値 *L-inst* より大きい状態でコミットが発生した場合，実行中のトランザクションの ID に対応するビットがセットされる．これらのフラグは前述した Stx-flags と同様にトランザクションを実行する毎に参照される．なお，前節で述べたように，トランザクションの実行命令数が少ないにもかかわらず，割り込み処理により誤って Ltx-flags がセットされる場合がある．そのような状況に対応するため，I-Counter の値が閾値 *S-inst* 以下の状態でコミットが発生した場合，実行中のトランザクション ID に対応するフラグをクリアする．

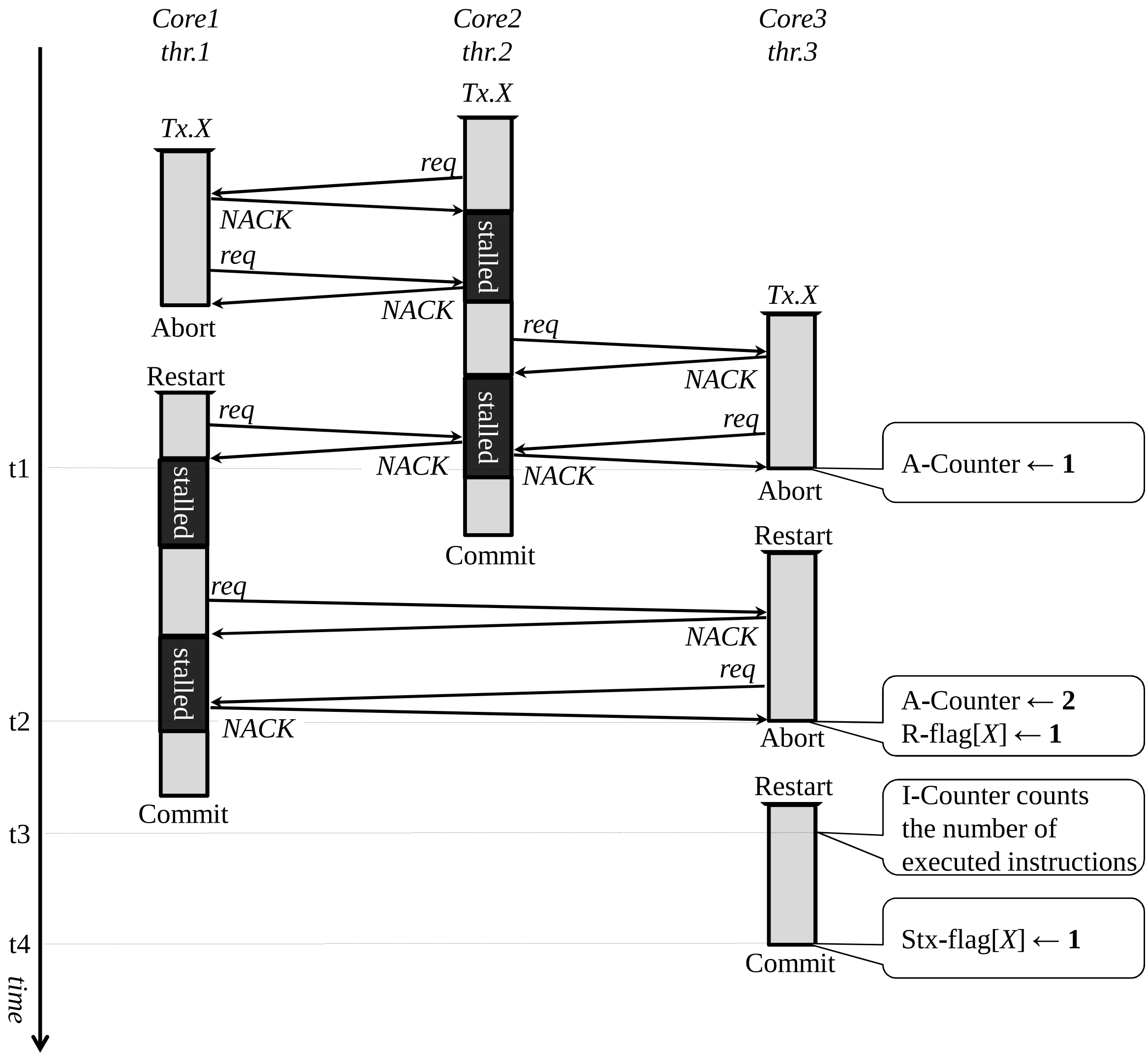


図 15: 逐次実行対象トランザクションの決定

競合が発生するトランザクションの実行例について，**図 15** を用いて *thr.3* における追加ハードウェアの動作を説明する．なお，説明を簡単にするため，閾値 $A\text{-}tx$ の値を 2 とする．

まず，*thr.3* の実行する *Tx.X* が *thr.2* との競合によりアボートされるとする（t1）．このとき，1 回目のアボートが発生したため，A-Counter の値は 1 となる．次に，*thr.3* は *Tx.X* を再実行し，今度は *thr.1* との競合により *Tx.X* をアボートするとする（t2）．このとき，A-Counter の値は 2 となり，閾値 $A\text{-}tx$ と等しくなる．ここで，R-flag のうち当該トランザクションの ID に対応するインデクスを持つビット R-flag$[X]$ の値をセットすることで，*Tx.X* がアボートを頻発させたトランザクションであることを記憶する．

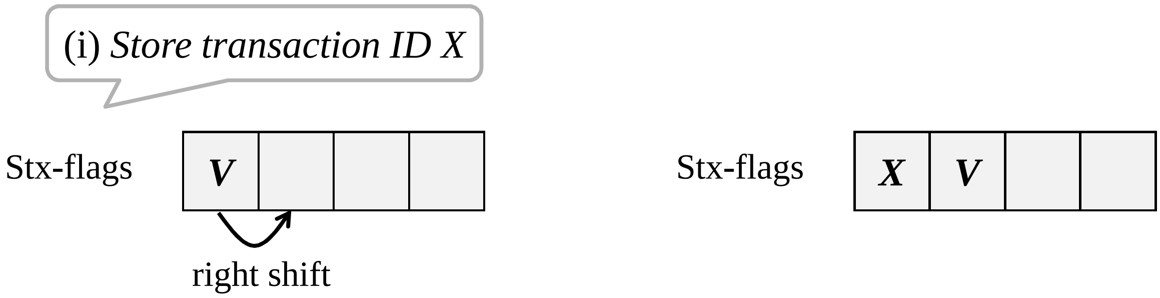
続いて *thr.3* は *Tx.X* を実行するが，R-flag[X] にビットがセットされているため，トランザクション内で実行される命令数が I-Counter でカウントされる．そしてこの I-Counter の値が，閾値 *L-inst* より小さい状態でトランザクションがコミットされるとき，実行するトランザクションの ID が X であるため，Stx-flag[X] のビットがセットされる（t4）．このようにして，*Tx.X* 中で実行された命令数が少なかったと記憶される．一方で，I-Counter の値が *L-inst* より大きい状態でトランザクションがコミットされる場合は，Ltx-flag[X] のビットがセットされる．その後，Stx-flag[X] のみがセットされている状態で *Tx.X* が実行される場合，このトランザクションは過去に競合を頻繁に引き起こし，かつ実行命令数が少ないと判定され，逐次実行の対象となる．

なお，一般に各コアでは複数のトランザクションが実行されるが，そのトランザクション数は不定であるため，Stx-flags および Ltx-flags は各コアに k ビットずつ用意することとする．なお，k 個より多いトランザクションを含むプログラムを実行する場合，フラグよりもトランザクションの数が多くなるため，各フラグがトランザクションの ID を 1 つ記憶するだけでは対応できなくなる．このような場合でも逐次実行対象とするトランザクションを正しく選択できる方法について検討する．まず，Stx-flags および Ltx-flags が保持する値はキューを用いて管理され，キューが一杯で新たに値を登録できない場合は最も古い値を追い出すこととする．値を格納するための各エントリ領域を固定長のビット列で構成した場合，この追い出す操作をシフト演算で実現することができる．なお一般的な HTM では，20 種類程度のトランザクションが設定できるようになっているため，各エントリ領域を 5 ビット長とすることで十分であると考えられる．

ここで，$k = 4$ の場合における Stx-flags の動作を**図 16** と**図 17** を用いて説明する．図 16(a) では，Stx-flags に既に値 V が登録されているとする．この状態でトランザクション ID X が登録される場合（i），登録済みの値 V が右シフトされ，最も左の位置に X が登録され，結果として図 16(b) のようになる．その後，図 17 のように，Stx-flags に空きがない状態でトランザクション ID Y が登録される場合（ii），さきほどの登録時と同様に登録済みの ID が右シフトされ，最も右に位置する ID は Stx-flags から追い出される（iii）．これにより，登録処理後の Stx-flags の状態は図 17(b) のようになる．また，Ltx-flags に対する操作も Stx-flags と同様に実現される．

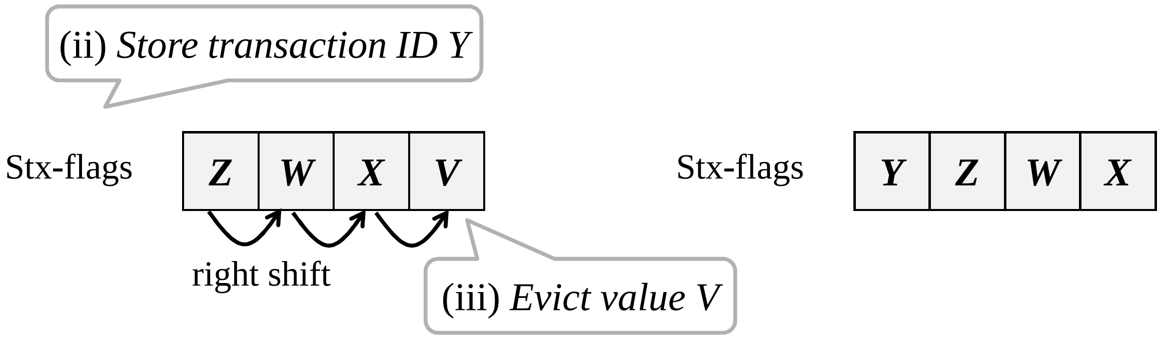
4.3.2　トランザクションの実行順序制御

前項で述べた逐次実行対象トランザクションの実行順序を制御するために，以下に示す 2 つの機構を各コアに追加する．また，コア数を n とした場合の追加機構を**図 18**



(a) 登録時　　(b) 登録後

図 16: Stx-flags へのトランザクション ID の登録



(a) 登録時　　(b) 登録後

図 17: 登録済みの ID が追い出される場合

に示す．

Sequential flag（S-flag）： 他スレッドが実行しようとしたトランザクションを待機させる必要があるかどうか判定する．このフラグは，自スレッドが逐次実行対象となるトランザクションの実行を試みた場合にセットされる．また，自身が実行するトランザクションのコミットを他のスレッドが待機し始めた場合にクリアされる．

ID of Opponent Thread（O-id）： 自スレッドが実行するトランザクションのコミットを待っている相手スレッドの番号を記憶する．この値はコミット時に参照され，どのスレッドが自身のコミットを待っていたかを知るのに用いられる．な

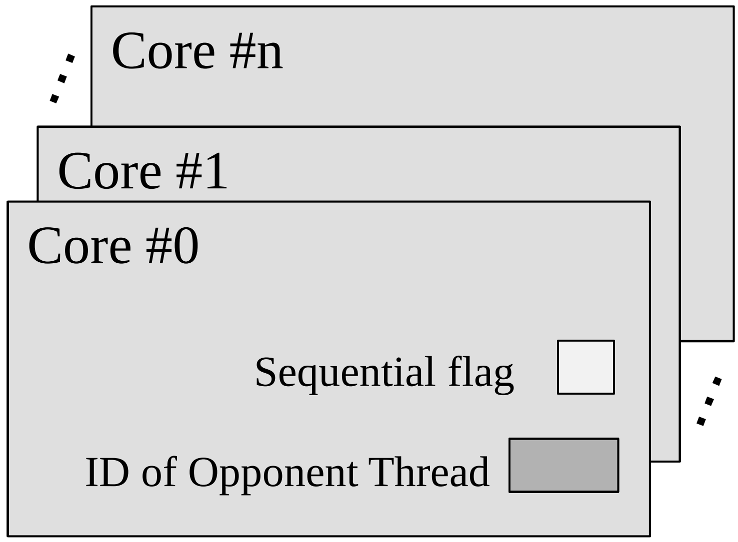


図 18: 実行順序を制御する機構

お，この値の参照により待機スレッドが特定された後，その待機スレッドに実行の許可が与えられるため，この値は参照後にクリアされる．

ここで，**図 19** の，逐次実行対象トランザクションの実行順序が制御される場合の例を用いて，3 つのスレッド（*thr.1*〜*3*）における追加ハードウェアの動作を説明する．なお，図 19 の $Tx.X$ は既に逐次実行の対象とされているとする．

まず，*thr.2* が逐次実行の対象である $Tx.X$ の実行を開始しようとした場合（t1），実行の許可を求めるリクエストが *thr.1* と *thr.3* に送信される．このリクエストを受信した *thr.1* および *thr.3* はそれぞれ S-flag を参照する（t2）．*thr.1* と *thr.3* は時刻 t1 までに逐次実行対象トランザクションの実行を試みていないため，S-flag はセットされていない．したがって，*thr.1* および *thr.3* から ACK が *thr.2* に返信される．この ACK を受信した *thr.2* は，$Tx.X$ の実行を開始するとともに，自身の S-flag をセットする（t3）．

続いて，*thr.1* が $Tx.X$ の実行を開始しようとした場合，さきほどと同様に実行の許可を求めるリクエストが他スレッドに送信される．なお，ACK が返信される場合のリクエストは省略している．このリクエストを受信した *thr.2* は，自身の S-flag がセットされていることから *thr.1* によるトランザクションの実行を待機させる必要があると判断する（t4）．したがって，*thr.2* は $NACK$ を *thr.1* に返信し，待機させた相手スレッドの番号である 1 を O-id にセットする．ここで，もし *thr.2* が実行するトランザクションのコミットをさらに別のスレッドが待機した場合，*thr.2* のコミット後に複数のスレッドが同時に待機状態から復帰してしまう．この復帰したスレッドはそれぞ

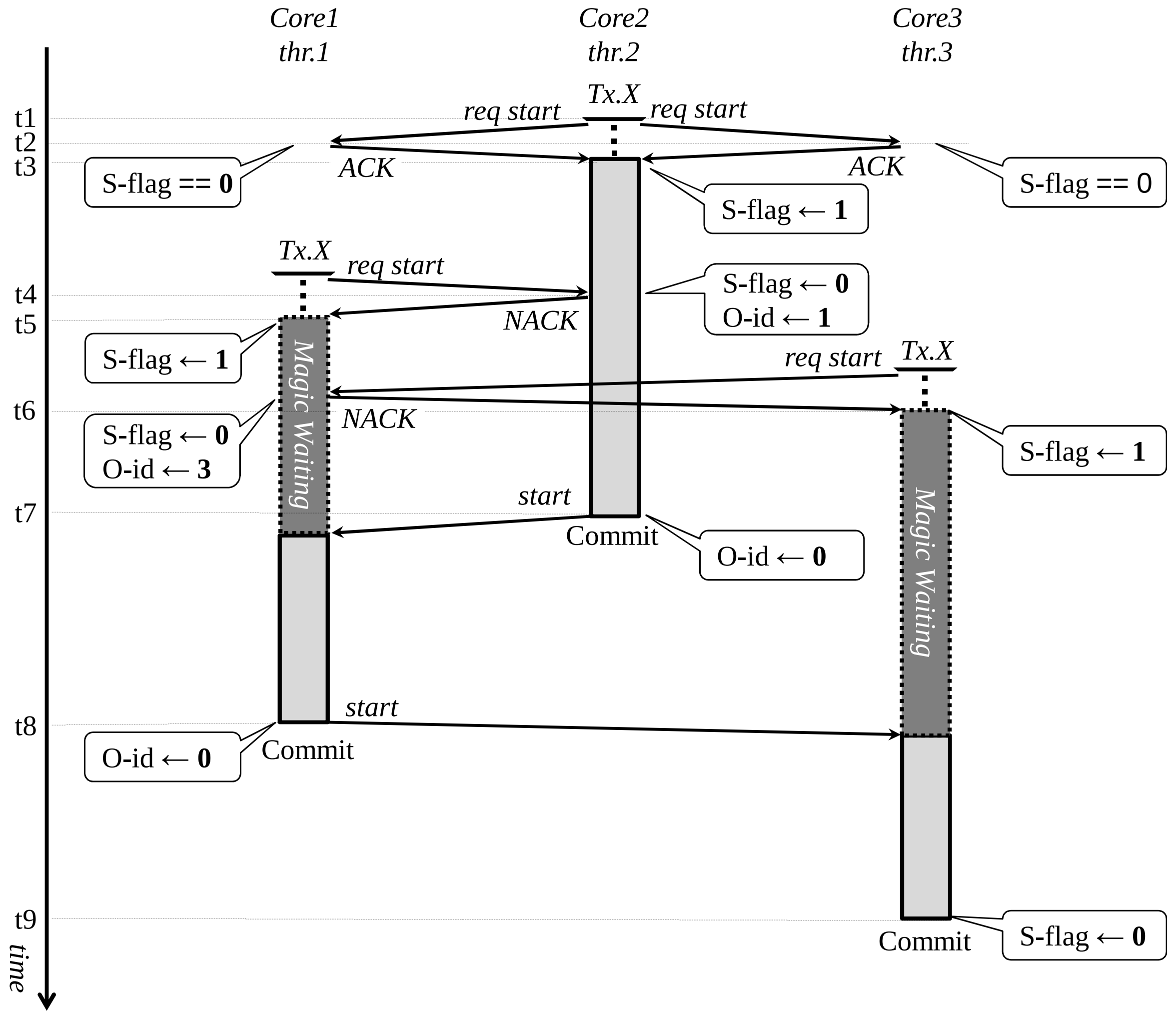


図 19: トランザクションの実行順序制御

れトランザクションの実行を開始するため，それらのスレッド間で競合が発生する可能性がある．そこで，*thr.2* は S-flag をクリアすることで自身のコミットを待機するスレッドを *thr.1* のみに限定する．

一方で *thr.2* から *NACK* を受信した *thr.1* は，自身の S-flag をセットし（t5），*thr.2* からトランザクションの実行許可が得られるまで待機し続ける．これにより，他のスレッドがトランザクションの実行を開始しようとした場合に *thr.1* のトランザクションがコミットするまで待機させることができる．次に *thr.3* が *Tx.X* の実行を開始しようとした場合も同様の処理が行われる（t6）．

その後，*thr.2* の実行する *Tx.X* がコミットされるとする（t7）．このとき，*thr.2* は O-id を参照することで，自身の実行するトランザクションがコミットされるのを待機

しているのは *thr.1* だということを知ることができる．したがって，*thr.2* は *thr.1* にトランザクションの実行許可を与える（t7）．また，*thr.2* のコミットを待機しているスレッドが存在しなくなるため，*thr.2* は自身の O-id をクリアする．続いて *thr.1* および *thr.3* の実行する *Tx.X* がコミットされる場合もさきほどのコミット時と同様の処理が行われる（t8，t9）．以上のようにして，トランザクションの逐次実行が制御される．

5　評価

3.3 節および 4.3 節で述べた拡張を，HTM の研究で一般的に用いられている LogTM[9] に実装し，シミュレーションによる評価を行った．本章ではまず，starving writer 解消手法の評価結果を示し考察する．そして，LogTM の他の競合解決手法との比較も行う．次に，futile stall 防止手法の評価結果を示し考察する．最後に starving writer 解消手法と futile stall 防止手法を組み合わせたモデルについての評価結果を示し考察する．

5.1　評価環境

評価には HTM の研究で広く用いられている Simics [10] 3.0.31 と GEMS [11] 2.1.1 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，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．プロセッサは 32 コアの SPARC V9 とし，OS は Solaris10 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 付属 microbench，SPLASH-2 [12]，および STAMP [13] から計 12 個を使用した．各プログラムに与えた入力パラメータを **表 2** に示す．

なお，各コアが 1 スレッドを実行するため，プロセッサ全体で 32 スレッド実行となるが，OS 用に 1 コアを使用するため，31 スレッドによる評価を行った．ただし，STAMP ベンチマークは 2 のべき乗数のスレッド数でしか動作しないため，STAMP に限り 16 スレッドで評価した．

5.2　Starving writer 解消手法の評価

図 20 および **表 3** に実行サイクル数比，**表 4** にアボート回数の削減率を示す．図 20 のグラフは既存の LogTM および starving writer 解消手法の 3 つのモデルの結果を示している．4 本のグラフはそれぞれ左から順に

(B)　既存の LogTM.

($\mathbf{S}_1$)　提案モデル 1：同一相手トランザクションとの WaR 競合による直近 2 回のア

表 1: シミュレータ諸元

Processor	SPARC V9
number of cores	32 cores
frequency	1 GHz
issue width	single-issue
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

ボートにおいて，そのアボートに関わったアドレスの組が一致した場合に magic waiting を有効にするモデル.

(S_2) 提案モデル 2：同一相手トランザクションとの WaR 競合による直近 2 回のアボートにおいて，そのアボートの直接の原因となったアドレスが一致した場合に magic waiting を有効にするモデル.

(S_3) 提案モデル 3：競合相手トランザクションを問わず，直近 2 回の WaR によるアボートにおいて，そのアボートの直接の原因となったアドレスが一致した場合に magic waiting を有効にするモデル.

の実行サイクル数比を表しており，既存手法 (B) の実行サイクル数を 1 として正規化している．また，凡例は内訳を示しており，Non_trans はトランザクション外，Good_trans, Bad_trans はそれぞれ結果的にコミット/アボートされたトランザクション内の実行サイクル数．Aborting，Backoff，Stall，Barrier，Magic_Waiting はそれぞれ，アボート，exponential backoff，ストール，バリア同期，magic waiting に要したサイクル数である．

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

表 2: ベンチマークプログラムの入力パラメータ

GEMS	
Btree	priv-alloc-20pct
Contention	config 1
Deque	4096ops 128bkoff
Prioque	8192ops
Slist	500ops 64len
SPLASH-2	
Barnes	512BODIES
Cholesky	tk14.0
Radiosity	-p 31 -batch
Raytrace	teapot
STAMP	
Genome	-g256 -s16 -n16384 -t16
Kmeans	-m40 -n40 -t0.05 -p16
Vacation	-n2 -q90 -u98 -r16384 -t4096 -c16

表 3: 実行サイクル数の削減率

		GEMS	SPLASH-2	STAMP	all
(S_1)	平均	5.3%	5.7%	1.3%	4.5%
	最大	8.4%	12.6%	1.9%	12.6%
(S_2)	平均	8.7%	10.2%	1.8%	7.5%
	最大	17.0%	18.6%	2.3%	18.6%
(S_3)	平均	8.5%	10.3%	1.7%	7.5%
	最大	17.3%	18.7%	1.9%	18.7%

ンを行うには，性能のばらつきを考慮しなければならない [14]．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表している．

評価に使用したベンチマークプログラムの多くは，starving writer 解消手法が解決すべき対象とした競合の再発，およびそれにともなうアボートの頻発を含んでいたた

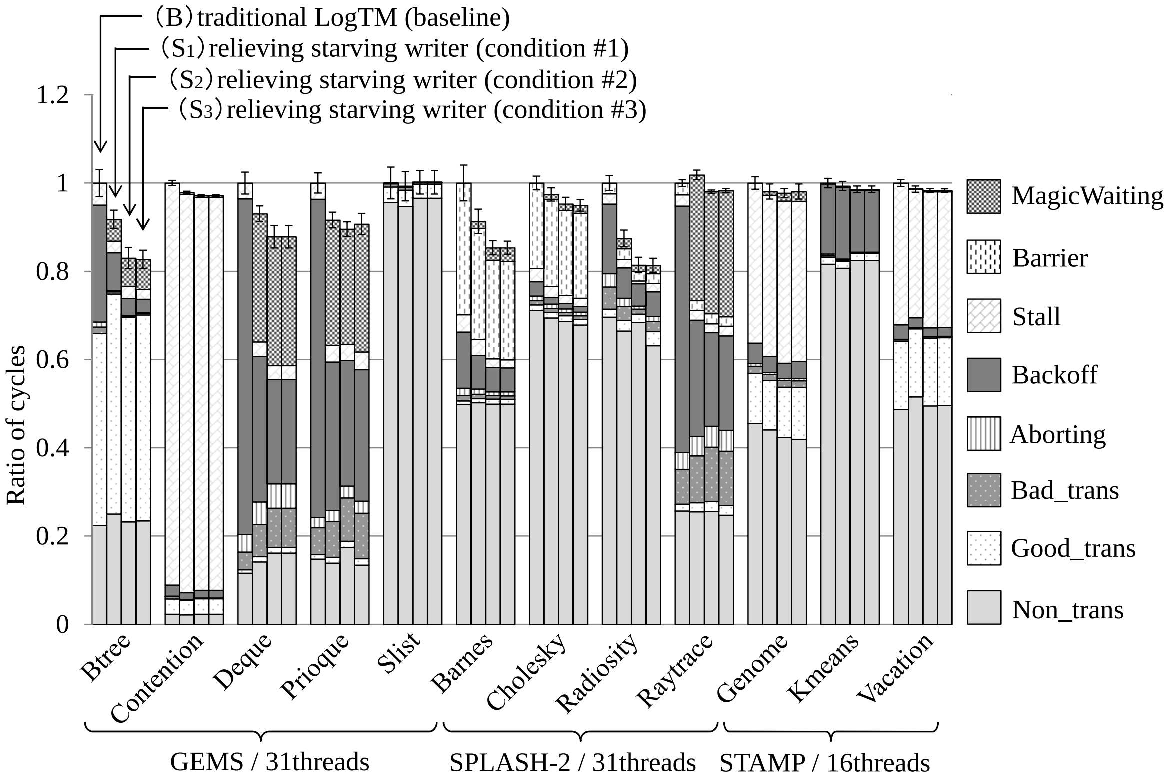


図 20: 実行サイクル数比（starving writer モデル）

表 4: アボート回数の削減率

		GEMS	SPLASH-2	STAMP	all
(S_1)	平均	38.8%	25.5%	40.0%	35.0%
	最大	76.2%	45.7%	67.7%	76.2%
(S_2)	平均	36.1%	44.7%	47.9%	46.1%
	最大	86.8%	67.1%	72.9%	86.8%
(S_3)	平均	35.7%	45.4%	47.6%	46.1%
	最大	86.6%	67.4%	72.9%	86.6%

め，提案手法によりこれらを解決することで性能が向上した．またアボートの発生回数に関しても，表 4 から分かるように大きく削減できており，提案手法が非常に効果的であることが分かる．なお，Slist は競合の繰返しがほとんど発生しないプログラムであるが，各提案モデルはそのような特徴を持つプログラムに対して性能に悪影響を及ぼさないため，既存モデルとほぼ同等の結果となっている．

プログラムを個別に見ると，まず Contention，Genome，Kmeans，Vacation では，

表 5: ストア待ちストールのサイクル数の平均値

	(B)	(S_3)	reduced
Btree	248,269	110,422	55.5%
Barnes	252,265	119,949	52.5%
Radiosity	239,285	173,493	27.5%

ほぼすべての手法で提案手法によりわずかに高速化している．これは主にアボートの抑制によるもので，アボート回数は既存手法 (B) に対し最大 72.9%（Kmeans），最低でも 17.1%（Genome）削減されている．また，全実行サイクルに占める magic waiting の割合は，たとえば Kmeans では 0.1%以下となっており，本提案によって新たに加えられた待機処理が短時間で済んでいることが分かる．しかしこれらのプログラムでは，元来アボートが実行サイクルに与える影響は小さかったため，高速化率は小さくなっている．

次に，Btree，Deque，Prioque，Barnes，Radiosity については，アボート回数の削減による Bad_trans や Aborting サイクルの減少，競合自体の削減による Stall サイクルの減少，アボートの繰返しを抑制したことによる Backoff サイクルの減少などにより大きく高速化しており，提案手法の有効性が確認できた．なおこれらのうち Deque および Prioque については Magic_Waiting の占める割合が比較的大きいことから，writer が早期にコミットできたというよりむしろ，magic waiting によって exponential backoff よりも適切な待ち時間が reader に設定された結果であると考えられる．しかし残り 3 つのプログラムについては，実行サイクル数全体に占める Magic_Waiting の割合が少ないため，writer が従来手法より早期にコミットに至ったことの効果が大きいと考えられる．またこれら 3 つのプログラムについて，ストールサイクル数の中でも特に，writer がストアを実行できずにストールさせられているサイクル数が (B) と (S_3) でどう異なっているかを調査した結果，**表 5** に示すように (S_3) では (B) に対し大きく削減できており，本提案手法の有効性が確認できた．

なお，これらの中でも Btree は最も starving writer が発生するプログラムであるが，starving writer 発生時のアボート抑制および Backoff 削減も高速化に寄与している．Btree において，手法 (S_2) および (S_3) のアボート回数を調査したところ，どちらの手法も既存手法に対し 86.8%も削減できていることが確認できた．また，最長のアボート繰返し回数についても，約 1/4 程度に削減されていた．

また，Barnes の場合，提案モデル (S_2)，(S_3) は，既存モデル (B) に対し Barrier を約 25%削減している．これは，既存モデルでは特定のトランザクションがアボートの繰返しにより遅延することで，バリア同期において他のトランザクションを長期間待機させてしまっていた状況を，提案モデルで解決できたためであると考えられる．

提案手法による効果が最も大きかった Radiosity では，提案手法により 30 スレッドが一時的に magic waiting を有効にする状況が見られた．これはすなわち，非常に競合を起こしやすいトランザクションが存在するということである．そして提案手法は，そのトランザクションの実行をコミットまで優先的に進行させることで，既存モデルで発生していたアボートの頻発を抑制したと考えられる．

一方，Deque，Prioque および Raytrace に見られる傾向として，アボート回数は減少しているものの Bad_trans サイクルが増加してしまっていることが挙げられる．既存手法ではトランザクション開始直後にアボートする状況が頻発しており，個々のアボートで計上される Bad_trans は少なかった．しかし提案手法では，トランザクション中のより多くの命令を実行した後にアボートする場合があり，アボート回数は少ないものの個々のアボートで計上される Bad_trans サイクルが増加してしまったためであると考えられる．

結果をまとめると，手法 (S_2)，(S_3) は (S_1) よりも高い性能を示しており，possible_cycle フラグをセットする原因となった競合アドレスを考慮せず，同アドレス競合によるアボートの繰返しを抑制することが重要であることが分かった．また，(S_2) と (S_3) には有意な差はなく，競合相手スレッド別に競合アドレスを記憶する必要性は低いと考えられることから，ハードウェアコストも軽量である手法 (S_3) が最も優れていると言える．

さて，HTM で用いられる代表的な競合解決手法は，2.3 項で述べたように possible_cycle フラグを用いてデッドロックを回避するものであり，本研究の提案手法もこれをベースとしている．一方，他の選択肢として，possible_cycle フラグを用いず，キャッシュリクエスト送信側トランザクションの開始時刻が受信側トランザクションの開始時刻よりも早かった場合に，即座に受信側トランザクションをアボートさせる方法がある．これを timestamp 方式と呼ぶ．

この方式を用いた場合，ストールがほとんど発生しないため，本研究で提案した手法と同様にトランザクションの starvation が抑制されると考えられる．しかし，必要以上にアボートが発生してしまうことで逆に性能に悪影響を及ぼす可能性があると考えられる．

そこでまず，timestamp 方式のモデル **(T)** について総実行サイクル数を調査し，pos-

表 6: アボートおよびストールに関わるサイクル数の比較

benchmark	program	(B)	(T)	(S_3)
GEMS	Btree	34.1%	9.7%	5.8%
	Contention	94.2%	193.9%	90.9%
	Deque	24.2%	17.2%	23.7%
	Prioque	84.2%	75.7%	46.8%
	Slist	0.9%	19.7%	0.4%
SPLASH-2	Barnes	19.5%	9.6%	9.0%
	Cholesky	8.3%	1.9%	4.8%
	Radiosity	26.1%	5.4%	10.9%
	Raytrace	70.1%	70.7%	40.6%
STAMP	Genome	43.1%	234.4%	42.2%
	Kmeans	16.8%	7.2%	14.4%
	Vacation	35.8%	425.9%	33.1%

sible_cycle フラグを用いる既存の LogTM モデル (B) と比較したところ，一部性能向上するプログラムも存在したものの，(B) に対して全プログラムの幾何平均で約 14.8%，算術平均で約 43.9%の性能低下を引き起こすことを確認した．

次にこの理由を調査するため，全体性能に悪影響を及ぼす要素であるアボートおよびストールに関わるサイクル数が，(B) と (T) でどのように異なっているかを調査した．その結果を**表 6** に示す．ここではまず，アボートに関わる Bad_trans，Aborting，Backoff とストールに関わる Stall の総和を，性能に悪影響を及ぼすサイクル数であると定義し，(B) においてこのサイクル数が総実行サイクル数に占める割合を表の (B) の列に示している．次に，timestamp 方式によって各ベンチマークプログラムを実行した結果から，上記 4 項目に要したサイクル数を (B) の総実行サイクル数で正規化した値を，(T) の列に示している．同様に，提案モデル 3 の結果も (S_3) の列に示した．

まず Btree，Barnes，Radiosity においては，(T) は (B) に対し，これらのサイクル数が (B) の総実行サイクル数比で約 10%以上減少しており，timestamp 方式が possible_cycle 方式よりも有利に働くプログラムが存在することが分かる．一方でこれらのサイクル数は，Contention で約 2 倍，Genome で約 5 倍，Vacation においては約 12 倍にまで膨れ上がってしまっており，これが原因でこれらのプログラムでは総実行サイクル数で

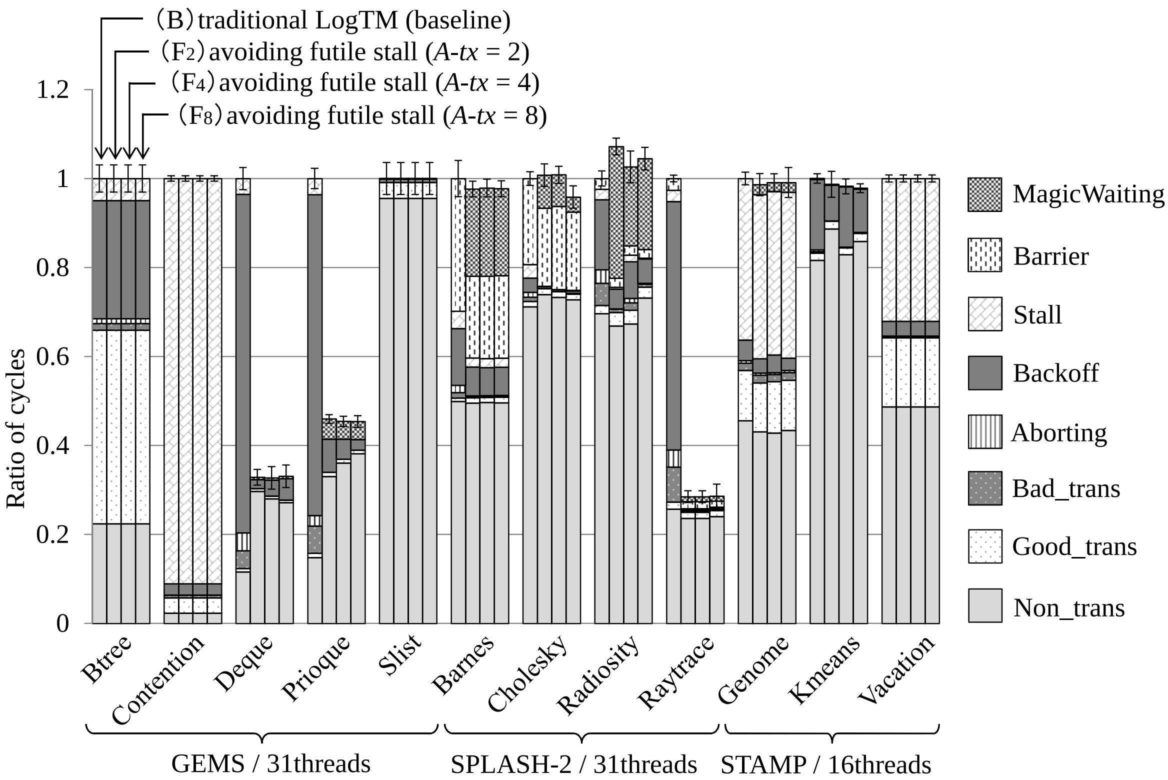


図 21: 実行サイクル数比（futile stall 防止モデル）

も (B) に対し，それぞれ約 2 倍，3 倍，4.5 倍を要していた．このように timestamp 方式では性能が著しく低下してしまうプログラムが存在することも分かった．

一方，提案モデル (S_3) では，(T) に有利となっていた Btree，Barnes，Radiosity で (T) と同等もしくはそれ以上の削減が実現できていることが分かる．また，(T) では大きく悪化してしまっていた Contention，Genome，Vacation においても，わずかながら (B) よりも改善されていることが分かる．さらに，(T) ではあまり改善できていない Prioque や Raytrace においても (S_3) は大きく改善できており，提案モデルの優位性が確認できた．

5.3 Futile stall 防止手法の評価

本節では，もう 1 つの提案手法である futile stall 防止手法の評価結果を示し考察する．**図 21** および**表 7** に実行サイクル数比，**表 8** にアボート回数の削減率を示す．この手法では，アボートを閾値 *A-tx* の回数だけ繰り返したトランザクションを逐次実行の対象とするが，閾値の変化に伴ってどのように性能が変化するかを調べるため，閾値を 2，4，8 とした場合の性能を評価した．4 本のグラフはそれぞれ左から順に

表 7: 実行サイクル数の削減率

		GEMS	SPLASH-2	STAMP	all
(F_2)	平均	31.5%	26.0%	0.9%	19.4%
	最大	67.1%	71.5%	1.3%	71.5%
(F_4)	平均	31.7%	26.8%	0.9%	19.8%
	最大	67.2%	71.5%	1.8%	71.5%
(F_8)	平均	31.6%	27.0%	1.0%	20.0%
	最大	66.9%	71.4%	2.2%	71.4%

表 8: アボート回数の削減率

		GEMS	SPLASH-2	STAMP	all
(F_2)	平均	89.1%	98.4%	54.8%	78.3%
	最大	99.6%	99.8%	88.8%	99.8%
(F_4)	平均	88.5%	98.1%	42.6%	71.8%
	最大	99.6%	99.8%	77.0%	99.8%
(F_8)	平均	85.9%	97.1%	34.1%	65.8%
	最大	99.3%	99.8%	65.0%	99.8%

(B) 既存の LogTM.

(S_1) 提案モデル 1：アボートを 2 回以上繰り返したトランザクションを逐次実行の対象とするモデル.

(S_2) 提案モデル 2：アボートを 4 回以上繰り返したトランザクションを逐次実行の対象とするモデル.

(S_3) 提案モデル 3：アボートを 8 回以上繰り返したトランザクションを逐次実行の対象とするモデル.

の実行サイクル数比を表しており，既存手法 (B) の実行サイクル数を 1 として正規化している．グラフ中の凡例は前節のグラフと同様であり，信頼区間は starving writer の評価と同様の手順で求めグラフ中にエラーバーで示している．なお，提案手法による待機処理に要したサイクル数は Magic Waiting として計上した．

また，3 つの提案モデルにおいて，トランザクション内での実行命令数の多寡を判定するために使用した閾値 *L-inst* および *S-inst* は以下の理由に基づいて設定した．ま

ず，競合が発生しやすく命令数が異なる 2 種類のトランザクション（Tx.X，Tx.Y）をそれぞれ逐次的に実行したところ，約 400 命令の Tx.X を逐次実行した場合には性能向上が得られ，約 900 命令の Tx.Y を逐次実行した場合には性能が低下したため，*L-inst* を 512 とした．また，逐次実行した際に性能向上が得られるトランザクションは 100 命令以下である場合が多く，OS による割り込みによってこれらのトランザクションの実行命令数が誤って *L-inst* より多いと判定された場合に対処するため，*S-inst* を 128 とした．

評価結果をプログラム別に見ると，まず Btree，Contention，Slist，Vacation については競合を引き起こしたトランザクションの実行命令数が閾値 *L-inst* よりも多く，これにより逐次的な実行が行われなかったため，既存モデルと同じ結果となった．次に，Cholesky と Radiosity について見ると，一部の提案モデルの性能がわずかに低下している．まず Cholesky は，アボートを起こしうるトランザクションが 5 種類存在するプログラムであり，(F_8) はこれらのうち 3 種類のトランザクションを逐次的に実行することで性能の向上を実現した．既存モデルでは，これら 3 種類のトランザクションは最大で 17 回もアボートを繰り返しており，(F_8) ではこのようなアボートが抑制されたことが性能向上につながったと考えられる．その結果，アボート回数の削減率は 93.7%と非常に高くなっている．しかし，(F_2) と (F_4) はさらにもう 1 種類のトランザクションを逐次的に実行したため，アボート回数の削減率はさらに向上したものの，総実行サイクル数は増加してしまった．これは，アボートが時折繰り返されるものの基本的には多く発生しないトランザクションを逐次実行したためである．

また，Radiosity はトランザクションが多く含まれているプログラムであり，各トランザクションのアボート繰り返し回数は様々であった。そのため，各モデルで逐次実行されたトランザクション数が異なり，これが性能の差につながった．まず，(F_2) では，ほとんどのトランザクションが逐次実行されてしまい，最も悪い結果となった．また，4〜7 回のアボートを繰り返すトランザクションが存在し，これを並列に実行した (F_8) の性能は (F_4) より悪いものとなった．これらの結果，各モデルの中では (F_4) において最も適切に逐次実行対象のトランザクションが選択された．しかしこのモデルにおいても，並列実行すべきトランザクションが逐次的に実行されたため，性能が (B) よりわずかに低下してしまった．また Barnes においても，同様の動作が発生したため，(B) に対する性能向上率がわずかなものになってしまった．

一方，Genome と Kmeans では各提案モデルで適切に逐次実行の対象が選択された．しかし，既存モデルにおいてそのトランザクションで発生していたアボートが性能に

表 9: Magic Waiting の割合

	(F_2)	(F_4)	(F_8)
Deque	0.5%	0.5%	0.5%
Prioque	4.5%	4.0%	4.0%
Raytrace	1.1%	1.1%	1.1%

与える影響は少なかったため，大きな性能向上が得られなかったと考えられる．

最後に，提案手法による効果が最も大きかった Deque，Prioque および Raytrace は，アボートを頻繁に引き起こすトランザクションもしくは全くアボートを引き起こさないトランザクションのどちらかしか存在しないプログラムであり，どの提案モデルもアボートを頻発させるトランザクションのみを逐次実行の対象としたため性能が大きく向上した．また，これら 3 つのプログラムについて，提案モデルによる magic waiting が既存モデルの実行サイクル数と比べどの程度の割合となっているのかを調査した結果，**表 9** に示すように各提案モデルはわずかな待機時間を設けるだけで大きな性能向上を実現できており，本提案手法の有効性が確認できた．

なお，Deque および Prioque の内訳を詳細に見てみると，どの提案モデルも既存モデルと比べて Non_trans が大きく増加したことが分かる．これら 2 つのプログラムでは，トランザクション外の処理の大部分を random 関数の実行が占めていた．そこでこの random 関数が，Non_trans 増大の原因に関係しているか調べるため，この関数を線形合同法によって乱数を発生させる関数に置き換えて評価した．その結果，既存モデルと提案モデルの Non_trans が同程度となったため，random 関数が各提案モデルに何らかの影響を与えていると考えられる．

結果をまとめると，(F_8) が最もよい性能向上率を示したが，各提案モデルで大きな差異は見られなかった．しかし一部のプログラムでは，閾値 *A-tx* を小さくした場合や大きくした場合に性能の悪化が見られた．したがって，この性能の悪化率が最も小さい (F_4) が最もバランスのとれたモデルであると考えられる．

5.4　2 つの提案手法を組み合わせたモデルの評価

本節では，5.2 節および 5.3 節でそれぞれ述べた 3 つのモデルのうち，最も良い結果を示したモデル同士を組み合わせたモデルの性能を評価する．**図 22** に実行サイクル数比を示す．図 22 の 4 本のグラフはそれぞれ左から順に

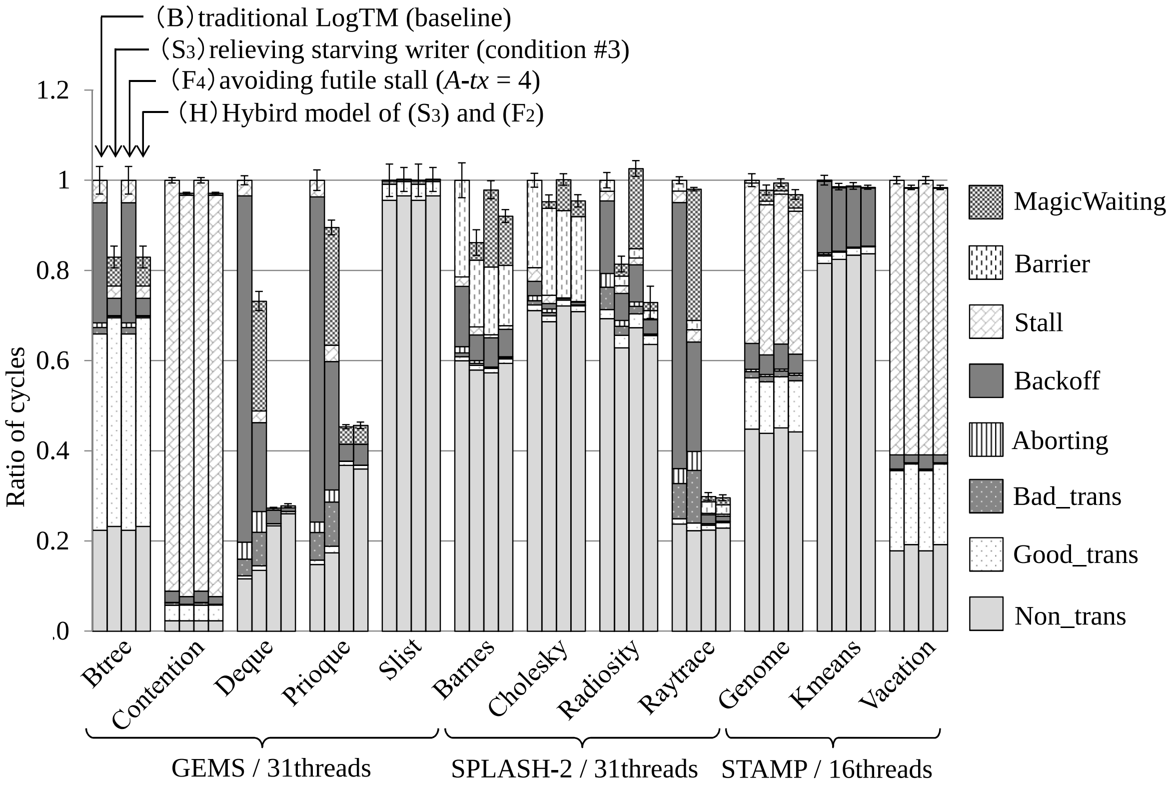


図 22: 実行サイクル数比（二つの手法を組み合わせたモデル）

(B)　既存の LogTM.

(S$_3$)　Starving writer 解消手法（モデル 3）：競合相手トランザクションを問わず，直近 2 回の WaR によるアボートにおいて，そのアボートの直接の原因となったアドレスが一致した場合に magic waiting を有効にするモデル.

(F$_4$)　Futile stall 防止手法（モデル 2）：アボートを 4 回以上繰り返したトランザクションを逐次実行の対象とするモデル.

(H)　Hybrid モデル：2 つの手法 (**S**$_3$)，(**F**$_4$) を組み合わせたモデル.

の実行サイクル数比を表しており，既存手法 (B) の実行サイクル数を 1 として正規化している.

評価結果を見てみると，(H) は Barnes 以外のプログラムにおいて，2 つの提案手法のうちの結果が良い方と同等の結果となったことが分かる．なお，Barnes では，(S$_3$) によって十分に高速化が得られる一部のトランザクションをプログラムの早い段階から逐次的に実行してしまったため，(S$_3$) ほど性能が向上しなかったと考えられる．一方 Radiosity と Genome では，(H) の結果が (S$_3$) および (F$_4$) の結果を上回った．これらのプログラムでは，starving writer が発生するトランザクションと競合が頻発する

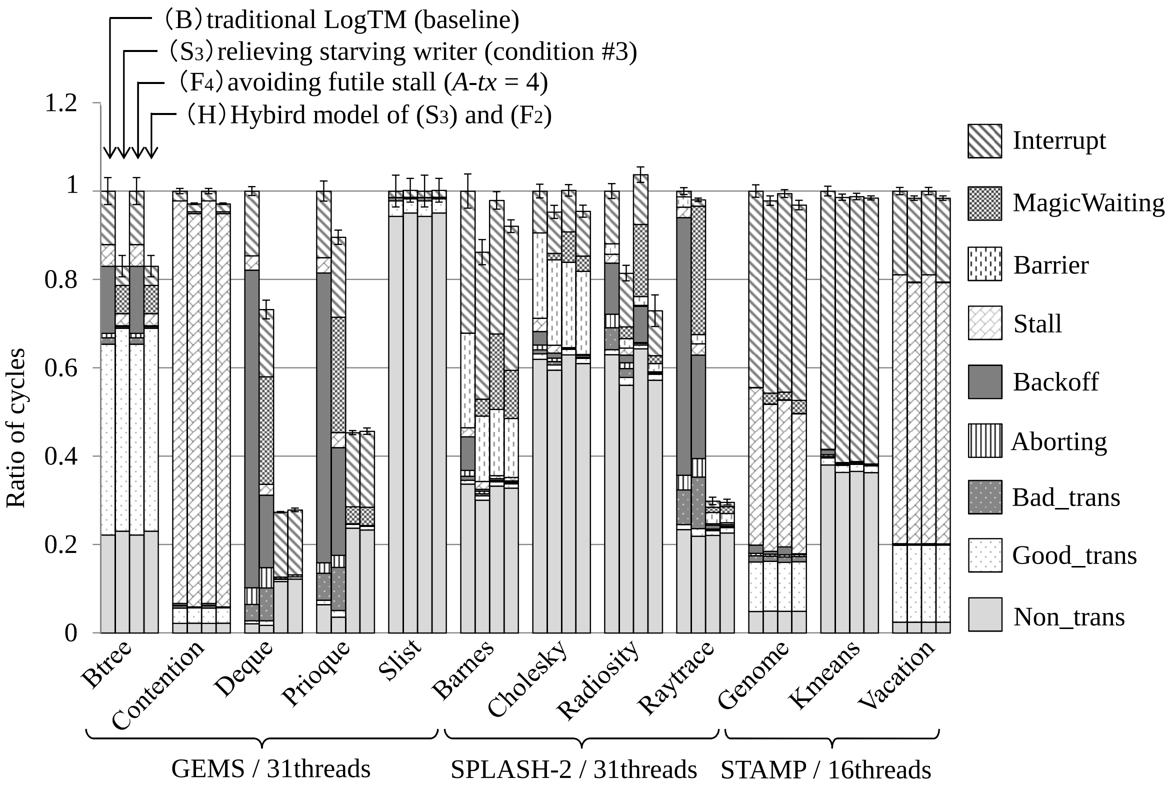


図 23: 実行サイクル数比（割り込み処理を別の内訳に計上）

トランザクションの両方が存在するため，トランザクション毎に適切な手法が適用され，(H) の性能向上率が最も高い結果になったと考えられる．

さて，これまでに示したグラフには，OS による割り込み処理が各内訳に含まれていた．したがって，この割り込み処理が提案手法の性能にどう影響するのか明確に示されていなかった．そこで，この影響をより細かく調査するため，この割り込み処理を，別の内訳 Interrupt として計上した場合の実行サイクル数比を**図 23** に示す．

まず，測定された割り込み処理の約 99%はトランザクション外の処理を実行中もしくは backoff 処理中に発生することが分かった．そのため，これらの処理中に特定のソフトウェア割り込み命令によって割り込みが発生したと考えられる．なお，Cholesky や Radiosity の内訳には Non_trans や Backoff サイクルが多く含まれているにも関わらず，Interrupt の割合が少ない結果になった．これは，このソフトウェア割り込み命令があまり実行されなかったことが原因である．

ここで，(B) の総実行サイクル数を 1 とした場合の各モデルにおけるこの Interrupt の割合を**表 10** に示す．表 10 から各モデルにおけるこの Interrupt の割合がほぼ同じであることが分かるため，割り込み処理は提案モデルの動作にあまり影響を与えないと

表 10: Interrupt の割合

		GEMS	SPLASH-2	STAMP	all
(B)	平均	9.1%	13.7%	40.6%	21.1%
	最大	15.0%	32.1%	58.4%	58.4%
(S_3)	平均	8.2%	14.0%	40.8%	21.0%
	最大	18.1%	33.2%	60.0%	60.0%
(F_4)	平均	9.4%	13.0%	41.2%	21.2%
	最大	16.8%	30.2%	59.9%	59.9%
(H)	平均	7.9%	13.5%	41.1%	20.8%
	最大	17.2%	32.6%	60.2%	60.2%

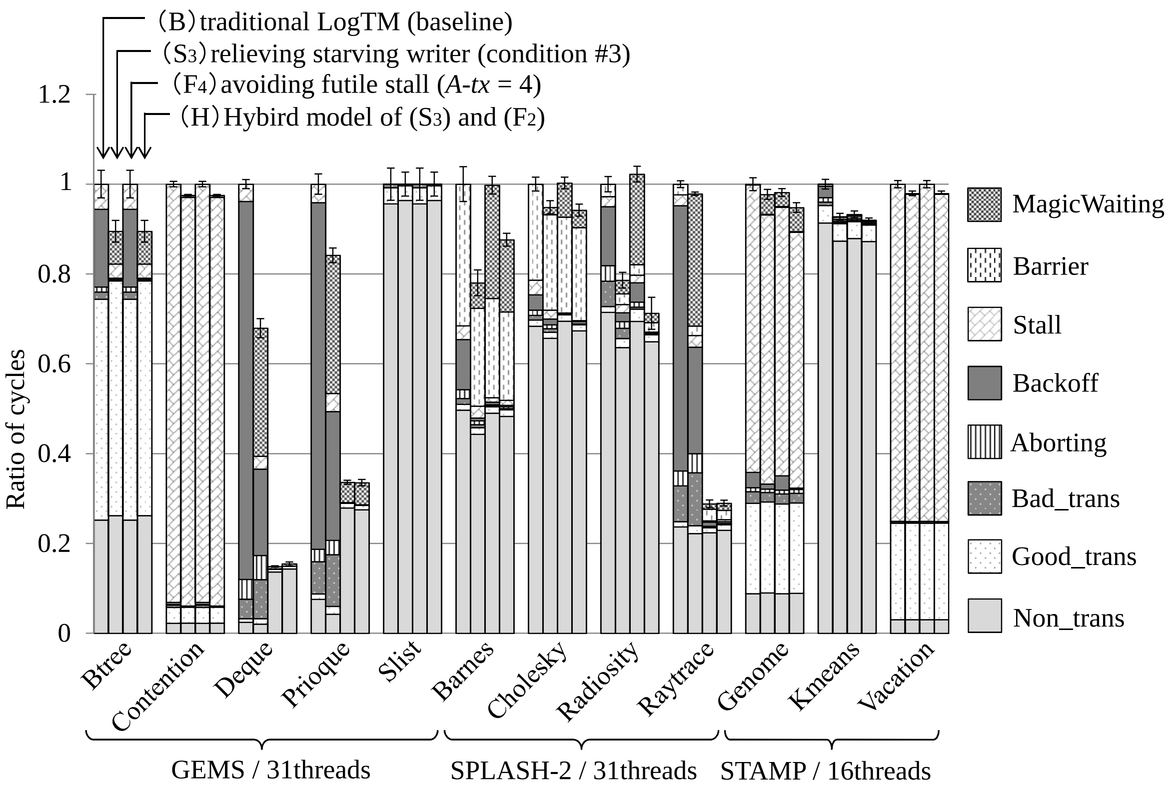


図 24: 実行サイクル数比（割り込み処理を除外）

考えられる．そこで，図 23 から割り込み処理を除いた場合の実行サイクル数比を**図 24** に示す．このグラフは，割り込み処理に要したサイクル数を除いた場合の既存モデルの実行サイクル数を 1 として正規化している．

図 24 を見てみると，Barnes の性能向上率のみが，図 22 で示した性能向上率よりも低下したことが分かる．まず Barnes で発生した割り込み処理について調査したところ，そのほとんどが Backoff 処理中に発生したものであった．そのため，この Backoff 処理を抑制した提案モデルでは，割り込み処理の割合が既存モデルより少ない結果となった．この理由から，割り込み処理を除いた Barnes の性能向上率は，図 22 で示した性能向上率よりも低下したと考えられる．しかし，他のプログラムでは全て (H) の性能向上率は図 22 で示した場合と同等かそれ以上となった．たとえば，Deque の結果を見てみると，(H) の性能向上率は 72.2%から 84.6% とさらに向上することから，プログラム本来の処理に対する提案手法の有効性が確認できた．

5.5　ハードウェアコスト

本節では，5.2 節および 5.3 節で評価した各提案モデルの H/W コストを見積もる．n スレッドを実行可能な n コア構成のプロセッサの場合，まず 3.2 節で述べた starving writer 解消手法の 1 つ目のモデルでは，1 コアあたりに必要となる WaR flags は n bit，また M-W flags は $2n$ bit であり，あわせて $3n$ bit と少量である．また C-Tbl については，幅 64 bit 深さ n 行の RAM で構成でき，たとえば $n = 32$ では C-Tbl サイズの総和は 16kB とやはり少量である．

なお，3.2 節で述べたように，starving writer 解消手法の 2 つ目のモデルの場合は 1 つのアドレスのみが条件に利用されるため，C-Tbl は 1 つあればよい．また，3.3 節で述べたように，このモデルでは M-W flags は不要である．モデル 1 では，主に 2 つの C-Tbl がハードウェアコストの大部分を占めており，このテーブルを 1 つに削減したモデル 2 のハードウェアコストはモデル 1 の約半分となる．そしてモデル 3 の場合は，競合相手ごとにアドレスが管理されないため，C-Tbl サイズの総和は 256B となる．このモデルにおけるすべての追加ハードウェアコストを合計すると 384B となることから，少量の追加ハードウェアにより starving writer 解消手法を実現できることが分かる．

一方，2 つ目の futile stall 防止手法では，1 コアあたりに必要となる A-Counter は，(F_4) の場合で 2 bit となる．また 5.3 節で評価した各モデルは，トランザクション内の命令数を 512 までカウントするため，I-Counter が 9 bit あればこれを実現できる．次に R-flags，Stx-flags および Ltx-flags は，4.3.1 項で述べたようにトランザクションの数が多いほど必要なコストが増える．今回使用したベンチマークには多くて 10 強のトランザクションが含まれていたため，これらのフラグはそれぞれ 16 bit あれば十分であると考えられる．なお，実際に競合を引き起こすトランザクションはさらに少ない

ため，これらのフラグはさらに少量で良いと考えられる．続いて，トランザクションの実行順序制御に必要な S-flag は 1 bit，O-id はスレッド数 n の場合に $\log_2 n$ となり，たとえば $n = 32$ では O-id のサイズは 5 bit となる．結果，(F_4) に必要なこれらのコストは，32 コア合計の場合 260B となる．以上のことから，どちらの提案モデルも少容量のハードウェアを追加することで実現できることが分かる．

6　関連研究

トランザクションのアボート後，その実行を最初からやり直すのではなく，途中から再実行することにより，その地点までの再実行を省略する部分ロールバック [15, 16, 17, 18, 19] に関する研究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 [20, 21, 22] など数多くの HTM に関する研究が行われてきた．前者の研究のように部分ロールバックを適用した場合，実行再開後，競合しやすいアドレスにアクセスする箇所まで到達するのに要する時間が短縮されるため，競合がより再発しやすくなる．一方で後者の研究のように競合検出やバージョン管理の方式を変更する場合，競合の起こり方を考慮していないため，本論文で取り上げたような競合パターンによって悪影響を受けてしまう．そこで本論文では，このような競合パターンを考慮しスレッドのスケジューリングを改良することで，従来の問題の解決を図った．

本研究と同様に，スレッドスケジューリングに着目した改良手法もこれまで数多く提案されてきた．Titos ら [23] は Eager 方式と Lazy 方式を組み合わせた新しい競合解決手法を提案した．彼らの手法では，基本的には Eager 方式が採用されている．しかし競合が発生した場合には，その競合対象のキャッシュライン上のデータを Lazy 方式のように扱う実行に切り替えることで，競合を引き起こしたデータを含むトランザクションのアボートを抑制した．

Yoo ら [24] は並列実行そのもののパフォーマンスを向上させるために，HTM に adaptive transaction scheduling（ATS）を実装することで，高い競合率によって並列性が欠落するようなワークロードのパフォーマンスを向上させる手法を提案した．ATS はアプリケーションからのフィードバックを用いて並列に実行するトランザクションの数を動的に制御することができる．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たが，ほとんどのベンチマークでは速度向上しておらず，提案手法の効果が得られたプログラムは一部であった．

Akpinar ら [25] 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

案した．それらのポリシーでは，アボートされたトランザクションの数などの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．さらに，彼らは既存のbackoffアルゴリズムよりも適切な待機時間を設定するアルゴリズムを考え，提案したポリシーと組み合わせることで競合の抑制を図った．しかし，性能向上が得られた手法は多くなく，最も性能向上率の高い手法でも最大で15%の速度向上しか実現できておらず，その効果は大きいものではない．

Geoffreyら[26]は複数のトランザクション内で共通してアクセスされるアドレス数に注目し，共通の度合をSimilarityと定義した．彼らはbloom filterを用いてこのSimilarityを見積もる手法を提案し，Similarityが一定の閾値を超えた場合，トランザクションを逐次的に実行することで競合の再発を抑制する．しかし，この手法によって，既存のHTMに対して速度性能がどの程度向上したのか示されていない．

Gaonaら[27]は，消費エネルギーを削減するための新たな手法を提案した．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，それらのトランザクションに優先度が設定され，逐次的に実行される．優先されたトランザクションが実行されている間，優先されなかった残りのトランザクションを待機させることで消費電力が抑制される．しかし，既存のHTMに対して消費電力の削減が10%しか実現できておらず，また速度性能は既存のHTMと同程度であった．

以上に述べた手法はいずれもアボートや競合発生の有無などの情報に基づいてスレッドの振る舞いを決定している．しかしこれらの情報を用いるだけでは，単に競合が発生しただけでもスレッドの振る舞いに変化が生じる可能性がある．また，スレッドの振る舞いを決める際に，著しく性能が低下する際の競合パターンに注目していないため，適切に競合を解決できない可能性もある．

一方，本論文は，性能に悪影響が及ぼされる場合のアクセスパターンやアボートが繰り返される動作に注目した．また，トランザクションのアボートが繰り返される場合，そのトランザクションは再びアボートされる場合が多く，この動作によりHTMの性能が大きく低下する可能性が高い．したがって，アボートの繰り返しを引き起こすような競合を抑制することが重要であると考え，これを解消することで性能向上を図る手法を提案した．

7　おわりに

本論文では，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の実行時に悪影響を及ぼす可能性のある競合パターンを2つ述べ，これらを解決するた

めの手法をそれぞれ提案した．1 つ目は，Write アクセスをしようとするトランザクションが飢餓状態に陥ったと考えられる場合，Read アクセスをしようとするトランザクションの実行を待機させることで，Write アクセス側のトランザクションを優先的にコミットする手法である．この手法により，実行の性能に悪影響を及ぼす starving writer が，既存の競合解決手法よりも早期に解消されることを確認した．2 つ目は，競合を頻繁に引き起こすトランザクションが存在すると考えられる場合，そのトランザクションを逐次的に実行する手法である．この手法により，アボートの繰り返しが抑制されることを確認した．なお，その対象トランザクションを決定する際に実行命令数を考慮することで，逐次実行の時間が長くなり過ぎることを防止した．

提案したこれら 2 つの手法の有効性を検証するために，既存の LogTM に提案手法をそれぞれ実装し，GEMS 付属の microbench，SPLASH-2，および STAMP の 3 種のベンチマーク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，2 つの手法を組み合わせたモデルでは競合再発に起因するアボートの繰り返しおよびストールを抑制することで，アボートによって結果的に破棄されるトランザクション内の実行サイクル数や，再実行までに設けられる待機処理である backoff サイクルが大きく削減されることを確認した．その結果，既存の LogTM に比べて実行サイクル数が最大で 72.2%，平均で 28.4%削減されることを確認した．また，これら 2 つの提案手法の各モデルを実現するにあたり必要なハードウェアコストを見積もった結果，各提案手法のうち最も性能の良いモデルのハードウェアコストはそれぞれ 384B，260B であり，少量のハードウェアを追加することでこれらの手法が実現できることを示した．

今後の課題として以下の 3 つが挙げられる．まず 1 つ目の課題として，starving writer 解消手法の改良である．たとえば，starving writer 解消手法に部分ロールバックを組み合わせることで，再実行コストを削減することが考えられる．その際，特定の命令の実行に優先度を設ける，もしくは backoff のための期間を改善することで，競合の再発を防止する必要がある．

次に 2 つ目の課題として，futile stall 防止手法でより適切に逐次実行対象のトランザクションを選択することが挙げられる．これを実現するために，futile stall 防止手法で用いた閾値 $A\text{-}tx$，$L\text{-}inst$ および $S\text{-}inst$ をプログラムに合わせて動的に設定することもしくは，別の指標を設けることなどが考えられる．

最後に 3 つ目の課題として，提案手法の適用による動作変化についてより詳細に考察することが挙げられる．たとえば，futile stall 防止手法の適用により一部のプログラムでトランザクション外の実行サイクル数が増加してしまったため，この原因を追

求することが考えられる．また，提案手法を適用した場合に電力消費量やコア間通信量がどのように変化するかを計測することで，提案手法が実行サイクル数以外にどのような影響を与えるのか調査することも挙げられる．

謝辞

本研究を進めるにあたり，研究の機会を与えて下さり，何度も貴重なご意見を賜わり，夜遅くまで相談に付き合って頂くなど，終始熱心にかつ丁寧に御指導頂いた名古屋工業大学の津邑公暁准教授に深く感謝致します．そして，本研究のために，多大な御尽力を頂き，御指導を賜わり，幾度となく貴重な助言を頂いた松尾啓志教授，齋藤彰一准教授，松井俊浩准教授，梶岡慎輔助教に感謝の意を表します．また，共同で研究を進め，多くの指摘や刺激を頂いた浅井宏樹氏，堀場匠一朗氏，橋本高志良氏，鈴木大輝氏には感謝の念に堪えません．最後になりましたが，本研究はもちろん，実生活の面でも多くの助言，協力，激励を頂いた研究グループ内のメンバー，ならびに松尾・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．

著者発表論文

論文

1. Shoichiro HORIBA, Hiroki ASAI, Masamichi ETO, Tomoaki TSUMURA, Hiroshi MATSUO: “Fine-Grain Conflict Management for Hardware Transactional Mermory Systems Employing Eager Version Management”, Proc. 4th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA2013)
2. 江藤 正通, 堀場 匠一朗, 浅井 宏樹, 津邑 公暁, 松尾 啓志: “Starving Writer の解消による LogTM の高速化”, 情処処理学会論文誌 コンピューティングシステム Vol.5, No.5(ACS40), pp.55-65 (Oct. 2012)
3. 堀場 匠一朗, 江藤 正通, 浅井 宏樹, 津邑 公暁, 松尾啓志: “デッドロック検出の厳密化による LogTM のアボート削減手法”, 情処処理学会論文誌 コンピューティングシステム Vol.5, No.5(ACS40), pp.43-54 (Oct. 2012)
4. 江藤 正通, 堀場 匠一朗, 浅井 宏樹, 津邑 公暁, 松尾 啓志: “競合の再発抑制による LogTM の高速化手法”, 先進的計算基盤シンポジウム (SACSIS2012) 論文集, pp.116–123, 優秀若手研究賞 受賞 (May. 2012)
5. 堀場 匠一朗, 江藤 正通, 浅井 宏樹, 津邑 公暁, 松尾啓志: “LogTM における依存関

係情報を用いたアボート抑制手法", 先進的計算基盤シンポジウム (SACSIS2012) 論文集, pp.108-115 (May. 2012)

報文

1. 鈴木 大輝, 江藤 正通, 橋本 高志良, 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, (to appear) (Mar. 2013)
2. 橋本 高志良, 鈴木 大輝, 堀場 匠一朗, 江藤 正通, 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, (to appear) (Mar. 2013)
3. 堀場 匠一朗, 浅井 宏樹, 江藤 正通, 浅井 宏樹, 津邑 公暁, 松尾啓志: "競合検出単位の再粒度化によるトランザクショナルメモリの高速化", 情報研報 (SWoPP2012), Vol.2012-ARC-201, No.1, pp.1–10 (Aug. 2012)
4. 堀場 匠一朗, 江藤 正通, 浅井 宏樹, 津邑 公暁, 松尾 啓志: "複数トランザクション間の競合を考慮した LogTM におけるアボート対象選択手法", 情報処理学会第 74 回全国大会論文集, Vol.1, pp.207–208, 学生奨励賞 受賞 (Mar. 2012)
5. 江藤 正通, 浅井 宏樹, 津邑 公暁, 松尾 啓志: "LogTM における適切な競合レベル選択による効率的ロールバック", 情処研報 (SWoPP2011), Vol.2011-ARC-196, No.22, pp.1–9 (Jul. 2011)

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 Blue Gene Solution Blue Gene/Q Application Development*, 1 edition (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] 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).

[10] 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).

[11] 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).

[12] 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).

[13] 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).

[14] 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).

[15] 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).

[16] 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).

[17] 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).

[18] 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).

[19] Vallejo, E., Galluzzi, M., Cristal, A., Vallejo, F., Beivide, R., Stenstrom, P., Smith, J. E. and Valero., M.: Implementing Kilo-Instruction Multiprocessors, *Proc. Int'l Conf. on Pervasive Services (ICPS'05)*, pp. 325–336 (2005).

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

[21] 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).

[22] 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).

[23] Titos, R., Negi, A., Acacio, M. E., García, J. M. and Stenstrom, P.: ZEBRA:A Data-Centric, Hybrid-Policy Hardware Transactional Memory Design, *Proc. Int'l Conf. on Supercomputing (ICS'11)*, pp. 53–62 (2011).

[24] 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).

[25] 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).

[26] 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).

[27] 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).