卒業研究論文

LogTMにおける
依存関係情報を利用した競合解決手法

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学 工学部 情報工学科
平成 20 年度入学　20115132 番

堀場 匠一朗

平成 24 年 2 月 8 日

LogTM における
依存関係情報を利用した競合解決手法

堀場 匠一朗

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，共有リソースへのアクセス制御にロックが広く用いられている．しかしロックを用いる場合には，デッドロックや並列性の低下といった問題がある．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適用したものであり，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競合として検出する．競合を検出した場合，片方のトランザクションの実行を停止する．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た場合，それらのトランザクションがお互いにストールさせ合い，デッドロック状態に陥る可能性がある．このような場合，片方のトランザクションの実行結果を破棄するアボートを行う．一方競合が検出されない場合，実行結果を元のメモリアドレスへ反映させる．これをコミットという．この TM をハードウェア上に実装したもののひとつに，LogTM がある．

LogTM は，possible_cycle flag と呼ばれるフラグを用いてデッドロックの発生を回避する．しかし，この手法は 2 つのトランザクション間における競合情報のみを用いてアボート対象を決定し，3 者以上のトランザクション間の依存関係を考慮していない．そのため，実際にはデッドロック状態にない場合でもデッドロックとして検出され，アボートが過剰に発生する可能性がある．

本論文ではこの問題を解決するために，トランザクションをアボートする条件を真にデッドロックが発生した場合のみとする手法を提案する．これによりアボートを抑制し，LogTM の実行の高速化を図る．また，デッドロックを検出可能にした上で適切なアボート対象を選択する 2 つの手法を提案する．

提案手法の有効性を検証するため，LogTM を拡張して提案手法を実装し，シミュレーションによる評価を行った．評価の対象として GEMS microbench および SPLASH-2 ベンチマークプログラムを用いて評価した結果，アボートの抑制により最大 31.5%，平均 7.3%の実行サイクル数を削減した．

LogTM における
依存関係情報を利用した競合解決手法

目次

1 はじめに

プログラムの実行を高速化する手法として，スーパスケーラのような命令レベル並列性 (Instruction-Level Parallelism: ILP) に着目したものが研究されてきた．しかしながらプログラム中の ILP には限界があり，命令レベルの並列化を行うだけではプロセッサの性能向上が頭打ちになりつつある．一方，半導体技術の向上によって集積回路の微細化が進み，単一コアの性能向上が図られてきた．しかし，消費電力の増大や配線遅延の相対的な増大という問題から，単一コアの性能向上による高速化は難しくなってきている．この流れを受け，単一チップ上に複数のプロセッサ・コアを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．

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制御する必要があり，その制御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する必要があり，また各プログラムで最適なロックの粒度を設定しなければ並列性の向上が困難である，したがって，ロックはプログラマにとって必ずしも利用しやすい機構ではない．

そこで，ロックを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) [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をメモリアクセスに適用したものであり，複数トランザクションにおける同一アドレスへのアクセスを競合として検出する．競合を検出した場合，片方のトランザクションの実行を停止する．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た場合，それらのトランザクションがお互いにストールさせ合い，デッドロック状態に陥る可能性がある．このような場合，片方のトランザクションの実行結果を破棄するアボートを行う．一方競合が検出されない場合，実行結果を元のメモリアドレスへ反映させる．これをコミットという．この TM をハードウェア上に実装したもののひとつに，LogTM [2] がある．

LogTM ではトランザクションが投機的に実行され，possible_cycle flag と呼ばれるフラグを用いてデッドロックの発生を回避する．しかし，この手法は 2 つのトランザクション間における競合情報のみを用いてアボート対象を決定し，3 者以上のトラン

ザクション間の依存関係を考慮していない．そのため，実際にはデッドロック状態にない場合でもデッドロックとして検出され，アボートが過剰に発生する可能性がある．

本論文ではこの問題を解決するために，トランザクションをアボートする条件を真にデッドロックが発生した場合のみとする手法を提案する．これによりアボートを抑制し，LogTM の実行の高速化を図る．また，デッドロック状態となった複数のトランザクションから，適切なアボート対象を選択する二つの手法を提案する．

以下，2 章では本研究の背景であるトランザクショナル・メモリ及び LogTM の概要について説明する．3 章では，LogTM の問題点及び提案手法の動作モデルについて説明し，4 章でその実装方法について説明する．5 章では提案手法を評価し，6 章で結論を述べる．

2　背景

本章では，本研究の対象となるトランザクショナル・メモリ及びそれをハードウェアで実現したシステムのひとつである LogTM について述べ，また，その関連研究について説明する．

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，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制御する必要があり，その制御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，ロックを用いたアクセス制御では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，プログラムごとに最適なロックの粒度を設定することは難しい．例えば，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．一方細粒度ロックを用いる場合では，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複雑となる．以上のような特徴は，ロックを用いたプログラム設計が困難である要因となっている．

そこでロックを用いない並行性制御機構である**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**トランザクション**として定義され，トランザクションは以下の 2 つの性質を満たす.

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に実行した場合と同じである.

アトミシティ(原子性):
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，部分的に実行されてはいけない.

以上の性質を保証するために，TM はトランザクション内のメモリアクセスを監視する．このとき，あるトランザクション内でアクセスされたメモリアドレスと，他のトランザクション内でアクセスされたメモリアドレスが同一であった場合，これを**競合**として検出する．競合を検出した場合，片方のトランザクションの実行を停止する．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た場合，それらのトランザクションがお互いにストールさせ合い，デッドロック状態に陥る可能性がある．このような場合，片方のトランザクションの実行結果を破棄する**アボート**を行う．トランザクションをアボートしたスレッドはメモリ及びレジスタの状態をトランザクション開始時点の状態に戻し，再実行する．この状態を復元する一連の処理を**ロールバック**という．一方競合が検出されない場合，実行結果を元のメモリアドレスへ反映させる．これを**コミット**という.

TM はこのように動作することで，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列性が向上するため，コア数に応じた性能のスケールが期待できる．また，プログラマは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.

TM で行われる，ストールやコミット，アボート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ことで実現されている．ハードウェア上に実装された TM はハードウェア・トランザクショナル・メモリ (HTM) と呼ばれる．一般的に HTM では，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同時に保持するために，片方をキャッシュ上に保持し，もう片方を別の領域に保持している．また，競合を検出及び解決する機構をハードウェアによってサポートしており，速度性能が高い．一方で，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)[3] では，TM 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，HTM のよう

なハードウェア拡張は必要ないが，オーバヘッドが大きい．

2.2 LogTM

本節では，HTM の一種であり本研究のターゲットとなる **Log-based Transactional Memoey (LogTM)** について述べる．また，この LogTM におけるデータのバージョン管理方法及び競合検出方法について説明する．

2.2.1 データのバージョン管理

TM における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，アクセスするデータの古いバージョンを保持し管理する必要がある．LogTM は，このバージョン管理を仮想メモリ領域を用いることで実現している．具体的には，**ログ**と呼ばれる仮想メモリ領域をスレッドごとに割り当て，トランザクション内のストア命令によって上書きされる前の値と，そのアドレスをこのログに退避する．一方，ストア命令の結果はメモリに書き込まれる．なお，あるトランザクション内で同じアドレスに対して複数回ストア命令が実行された場合，ログに保存するのは最初の 1 回だけでよい．なぜなら，トランザクションをロールバックするには，トランザクション開始時点でのメモリ状態が分かれば十分だからである．以上で述べたバージョン管理を，あるスレッドがトランザクションを投機実行する例である，**図 1** を用いて説明する．図中の Thread，Memory 及び Log はそれぞれトランザクションを実行するスレッド，主記憶，割り当てられたログ領域を表す．図 1(a) はトランザクションの開始時点の様子を示しており，メモリの 0x100 番地には値 10 が格納されている．この状態からトランザクションの実行が進み，図 1(b) に示すように `ST 0x100 15` が実行されると，ストアアクセスされたメモリアドレスである `0x100` と，書き換え前の値である 10 が主記憶からログに退避され，ストアの結果である 15 が主記憶に上書きされる．

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には，図 1(c) に示すようにトランザクションをコミットする．このとき，ストア結果である値 15 は既に主記憶に保存されているため，ログの内容を破棄するだけでコミット操作を実現できる．

一方で投機的実行が失敗した場合，図 1(d) に示すように，トランザクション内で変更があった値を全て破棄し，トランザクションをアボートする．また，アボート後に実行を再開するには，トランザクションを開始状態までロールバックする必要がある．これを実現するため，LogTM ではトランザクションの開始時にその時点でのレジスタ状態等を保存し，その状態をチェックポイント (以下 CP) として登録しておく．そして，

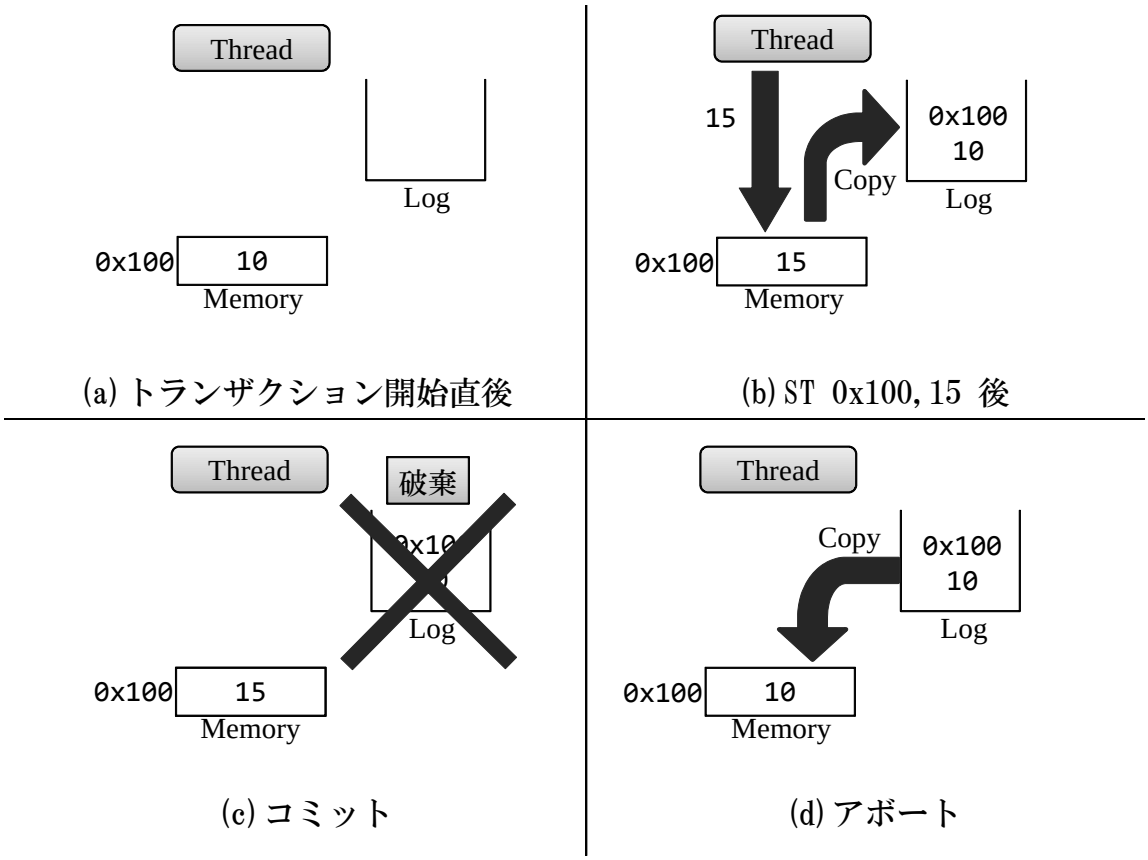

図 1: LogTM におけるバージョン管理

ロールバック時に CP を用いることで，トランザクション開始時点の状態を復元する．

このように，トランザクションをコミットする際にはメモリアクセスは全く行われない．一方，トランザクションをアボートする際にはログに保存された値を全て主記憶に書き戻す必要があるため，ログサイズに比例してメモリアクセスが増加する．LogTM はこのようにして，必ず行われるコミット操作を高速化することでプログラム全体の実行速度を高めようとしている．

2.2.2 競合検出

トランザクションのアトミシティを保つためには，あるトランザクション内のメモリアクセスと他のトランザクションのメモリアクセスとの間に競合が発生す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，LogTM ではキャッシュライン上に read ビット及び write ビットと呼ばれるフィールドを追加している．トランザクション内でリードアクセス，及び write アクセスが発生すると，ア

クセスのあったラインの read/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及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，LogTM ではキャッシュの一貫性を保持するキャッシュ・コヒーレンス・プロトコルを拡張している．LogTM では一貫性保持のモデルにディレクトリベース [4] の Illinois プロトコル [5] が採用されている．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，キャッシュラインの状態を変更させる要求を他の各スレッドに送信する．これをキャッシュ・コヒーレンス・リクエストという (以下リクエストと呼ぶ)．拡張したキャッシュ・コヒーレンス・プロトコルにおいて，各スレッドは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，キャッシュに追加された read ビット及び write ビットを参照する．これにより，各スレッドがトランザクション内であるメモリアドレスにアクセスしようとした場合，そのアドレスが他の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:
あるトランザクション内でライト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リード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リードアクセスリクエストがキャッシュ・コヒーレンス・プロトコルにより検出された場合である．このとき，トランザクション内で変更された値をコミットする前に他のトランザクションからアクセスされることになるため，トランザクションの性質を満たさない．

write after read:
あるトランザクション内でリード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ライト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ライトアクセスリクエストがキャッシュ・コヒーレンス・プロトコルにより検出された場合である．このときトランザクションの実行中であるにもかかわらず内部でアクセスした値が変更されることになるため，トランザクションの性質を満たさない．

write after write:
あるトランザクション内でライト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ライト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ライトアクセスリクエストがキャッシュ・コ

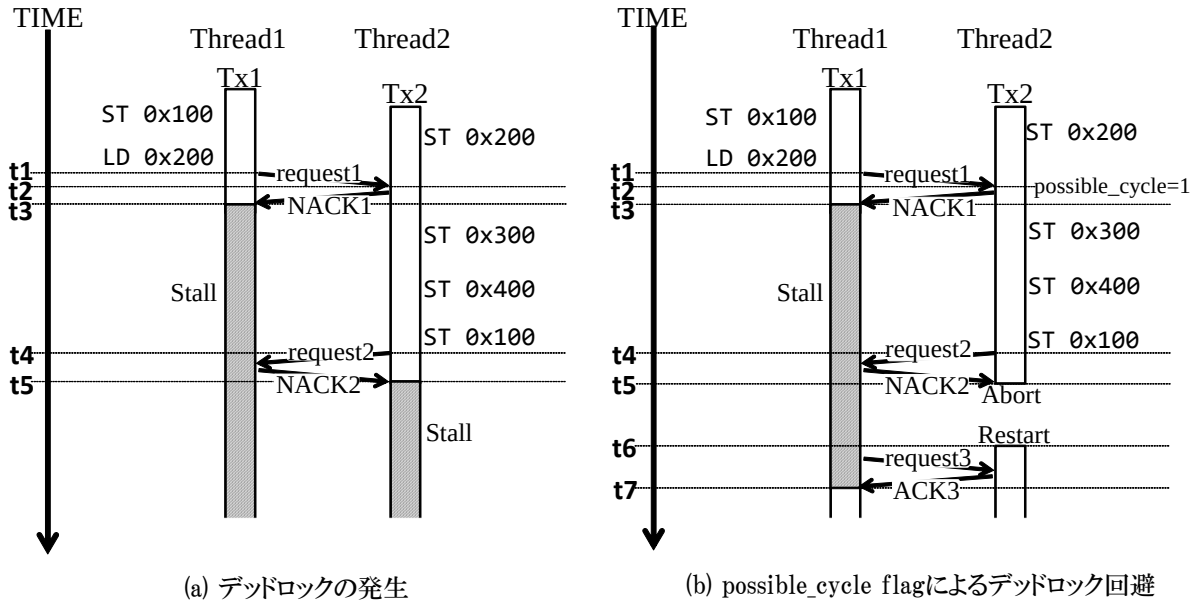

図 2: LogTM におけるトランザクションの競合解決

ヒーレンス・プロトコルにより検出された場合である．このときトランザクションの実行中であるにもかかわらず内部で書き込み対象アドレスの値が既に変更されていることになるため，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，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．一方で競合が発生しなかった場合は ACK が返信される．実際にはキャッシュの状態を一括管理しているディレクトリから返信されるが，便宜上メモリアクセスを行ったスレッドから送信しているものとして説明する．NACK を受信したスレッドは競合が発生したことを知り，競合したトランザクションが終了するまで一時的に実行を停止する．これを**ストール**という．トン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．競合したスレッドがその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，トランザクションをストールさせていたスレッドは相手の終了を検知し，実行を再開できる．しかし**図 2**(a) で示すように，複数のスレッドがトランザクションをストールさせるとデッドロック状態に陥る場合がある．この例では，2 つのスレッド Thread1 と Thread2 がそれぞれトランザクション Tx1 と Tx2 を投機的に実行している．まず，Thread1 が Tx1 の実行を開始した後に Thread2 が Tx2 の実行を開始しており，Thread1 が `ST 0x100` を，その後に Thread2 が `ST 0x200` を

実行済みである場合を考える．ここで，Thread1 が `LD 0x200` を実行しようとする場合，Thread1 は Thread2 に対して request1 を送信する (時刻 t1)．request1 を受信した Thread2 は競合したことを検知するため NACK1 を返信し (時刻 t2)，NACK1 を受信した Thread1 は Tx1 をストールさせる (時刻 t3)．図中では省略しているが，Thread1 はアクセスの許可を受けるまで Thread2 に対して定期的に request を送信している．この後，Thread2 による `ST 0x300`，`ST 0x400` の実行を経て `LD 0x100` が実行される (時刻 t4) と，Thread2 は Thread1 と競合し，Tx2 をストールさせる (時刻 t5)．このように，複数のスレッドが自身の実行する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お互いに待ち続けると，デッドロックに陥る．

このデッドロックを回避するために，LogTM では各スレッドに possible_cycle flag を保持させている．possible_cycle flag は Transactional Lock Removal[6] の分散タイムスタンプに倣った方法を採用しており，図 2(b) のように，自身より早くトランザクションを開始したスレッドに NACK を返信するとセットされる (時刻 t2)．そして，possible_cycle flag がセットされているスレッドは，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防ぐために自身の実行するトランザクションをアボートさせる (時刻 t5)．このようにして，開始時刻のより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る．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状態を復元し，Tx2 を再実行する (時刻 t6)．また，Thread2 がトランザクションをアボートしたことで Thread1 は 0x200 番地にアクセスできるようになるため，Tx1 をストール状態から復帰させる (時刻 t7)．

2.3 関連研究

トランザクションの途中から再実行することにより，その途中地点までの実行を省略する部分ロールバックに関する研究 [7] や，適切なスレッド数を動的に設定する研究 [8] など数多くの LogTM に関する研究が行われている．前者の手法を改良した伊藤らの研究 [9] では，競合を起こした命令のプログラムカウンタの値を記憶し，再度そのプログラムカウンタに該当する命令を実行する際，その位置を CP として設定することで，再実行命令数の削減を可能にしている．また，既存の LogTM では CP の作成数に制限があったが，その制限を無くす手法も提案している．しかし，LogTM で標準的な possible_cycle flag を用いる方法ではなく，競合発生時に即座にトランザクションがアボートする，よりアボートが発生しやすいベースラインモデルに対する高速化

で評価している．また，評価に対する考察が少なく，改良モデルのベースとしている Williullah らによる手法 [10] との比較評価もなされていないため，提案手法による効果の範囲が明らかになっていない．

一方後者のスレッド数の動的制御に関する研究では，競合とトランザクション数に相関関係があることに着目し，動的にスレッド数を調整することでアボート数を削減し，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ーヘッドや，実装に必要なハードウェアコストについて評価していない．また，評価に用いたベンチマークプログラムが一つのみであり，効果の汎用性も明らかではない．

3　アボート抑制手法の提案

本章では，既存手法である LogTM の問題点と，それを解決する提案手法について説明する．

3.1　問題点

2.2.2 項で述べたとおり，LogTM では possible_cycle flag を用いてアボート対象を選択する．しかし，この手法はある特定の 2 者間における競合情報のみを用いてアボート対象を決定し，3 者以上のトランザクション間の依存関係を考慮していない．したがって，実際にはデッドロックを起こしていない場合にもアボートが発生してしまう．

ここで，**図 3** 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が possible_cycle flag を利用して競合を解決する様子を示す．この例では，3 つのスレッド Thread1，Thread2，Thread3 がそれぞれトランザクション Tx1，Tx2，Tx3 を投機的に実行している．ここで，Thread1，Thread3 がトランザクションの実行を開始した後に Thread2 が Tx2 の実行を開始し，Thread1 が `ST 0x100` を，Thread3 が `LD 0x300` を実行後，Thread2 が `ST 0x200` を実行済みである場合を考える．まず，Thread1 が `LD 0x200` を実行しようとする場合，Thread2 へ request1 を送信する (時刻 t1)．ここで Thread2 は競合を検出し，自身より早くトランザクションを開始した Thread1 へ NACK1 を返信するため，possible_cycle flag がセットされる (時刻 t2)．この後，Thread2 が `ST 0x300` を実行しようとする場合，Thread3 へ request2 を送信するが，Thread3 は既に当該アドレスへのアクセスを行っているため競合を検出し，Thread2 へ NACK2 を返信する．このとき，Thread2 は possible_cycle flag がセットされている状態で，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 を受信するため，自身のトランザクション Tx2 をアボートする（時刻 t4）．このようにして，possible_cycle flag をセット

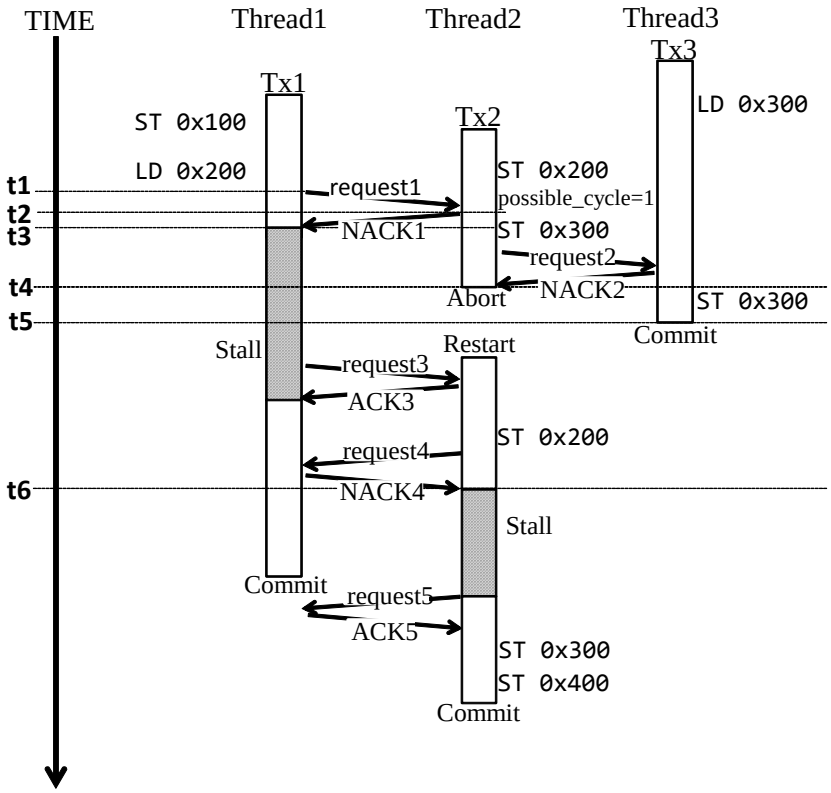

図 3: possible_cycle flag による競合解決の問題点

する原因となった Thread1 以外の相手である，Thread3 の実行する Tx3 に影響を受け，実際にはデッドロックを起こしていない場合にも Thead2 は Tx2 をアボートさせてしまう．この後，Thread3 は Tx3 をコミットする (時刻 t5)．一方 Tx2 をアボートした Thread2 は再実行を開始するが，時刻 t6 において Thread1 と競合するため，NACK を受信する．さらにその後，Thread1 がトランザクションをコミットすることで Thread2 は実行を再開できる．

3.2 アボート条件の厳格化

スレッド数の増加に伴い，並列実行されるトランザクション数が増加すると，3.1 節に示す無駄なアボートはより多く発生するようになると考えられる．したがって，このようなアボートの発生は抑制されることが望ましい．そこで本論文では，トランザクションのアボート条件を真にデッドロックが発生した場合のみとする手法を提案する．これによりログの書き戻し処理や再実行のオーバヘッドを削減し，LogTM の高速

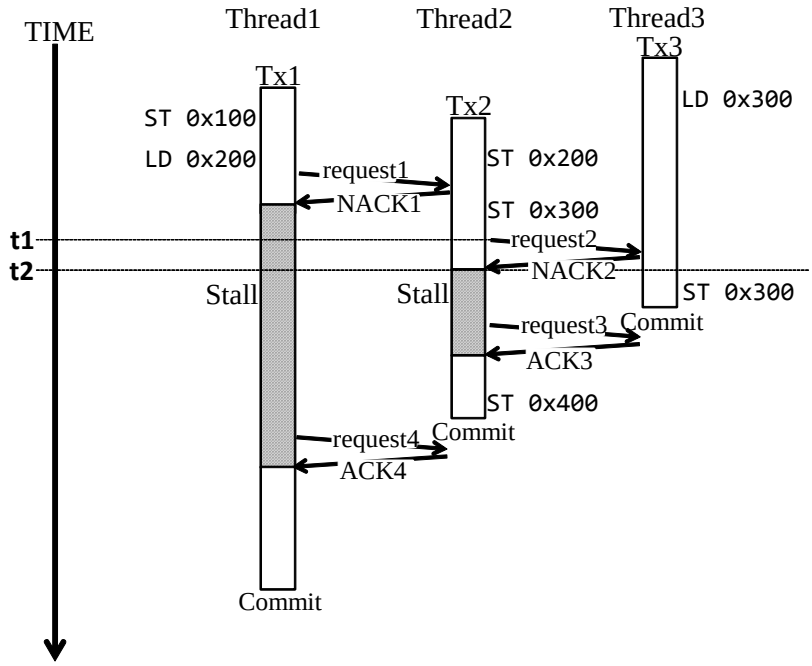

図 4: アボート条件を厳格化した動作モデル

化を図る．図 3 と同じ例に提案モデルを適用した場合の動作を**図 4** に示す．まず，時刻 t1 において Thread2 が `ST 0x300` を実行しようとする場合，Thread3 は既に当該アドレスへのアクセスを行っているため競合を検出し，Thread2 へ NACK2 を返信する．このとき，Thread2 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 を受信するが，自身の実行する Tx2 をアボートせず，ストール状態へと移行させる (時刻 t2)．さらにその後，各スレッドは自身と競合したスレッドがトランザクションをコミットするのを待って，順次実行を再開する．

提案モデルでは，このようにしてトランザクションのアボートを抑制することで，ログの書き戻し及び再実行に要するオーバヘッドを削減する．一方，Thread1 のように一部のトランザクションでストールサイクルが増大することで並列実行スレッド数が減少し，性能が悪化する可能性もある．具体的なデッドロック検出方法については，4 章で説明する．

3.3 アボート対象の選択

デッドロックを解決する最も簡単な方法は，デッドロックを検出したスレッドが，自身の実行するトランザクションをアボートさせることである．また，デッドロックを

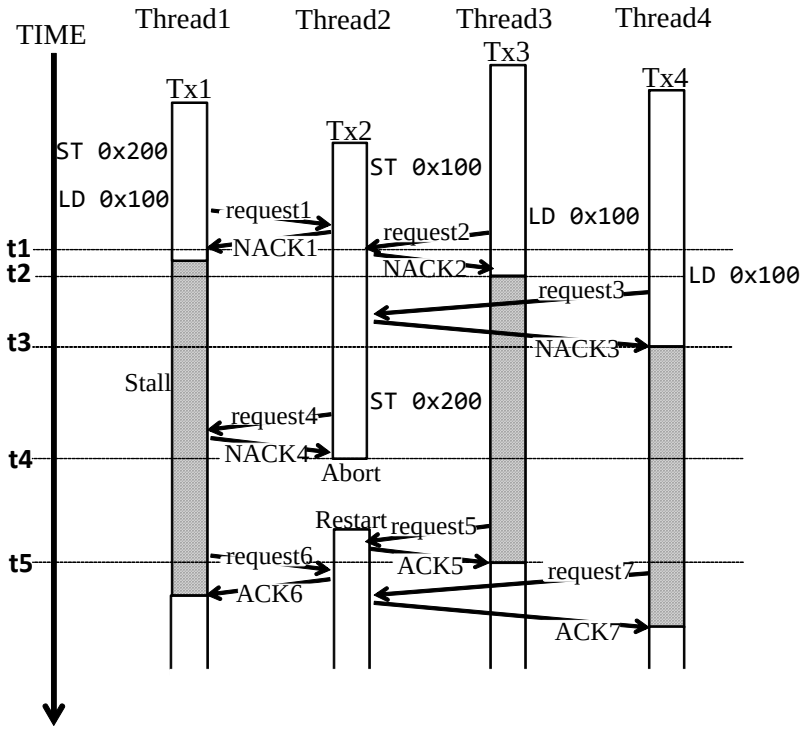

図 5: スレッドの競合相手数比較による選択

引き起こしているトランザクションの中から適切なアボート対象を選択することができれば，さらなる性能向上が期待できる．そこで，各スレッドの競合相手数及び実行するトランザクションの開始時刻に着目した，アボート対象を選択する二つの手法を提案する．

3.3.1 スレッドの競合相手数を比較

まず一つ目は，デッドロック検出時，自身がストールさせているトランザクションの数が，最も多いスレッドの実行するトランザクションをアボート対象とする手法である．これによって，より多くのスレッドがトランザクションの実行を再開でき，並列実行スレッド数が増加すると考えられる．ここで，あるトランザクションが，同一のアドレスに対してアクセスした複数のトランザクションをストールさせる様子を**図 5** に示す．この例では，4 つのスレッド Thread1，Thread2，Thread3，Thread4 がそれぞれトランザクション Tx1，Tx2，Tx3，Tx4 を投機的に実行している．

まず，Thread1 が `ST 0x200` を，Thread2 が `ST 0x100` を実行済みである場合を考える．この後，Thread1，Thread3，Thread4 が `LD 0x100` を実行しようとする場合，Thread2 では 3 つのスレッドとの競合が検出されるため，それぞれに NACK が返信

される．そして，NACK を受信した各スレッドは自身の実行するトランザクションをストールする (時刻 t1,t2,t3)．その後，Thread2 は `ST 0x200` を実行しようとするが，Thread1 は競合を検出するため，リクエストに対して NACK4 を返信する．このとき，Thread1 と Thread2 の間でデッドロックが発生する (時刻 t4)．ここで，Thread1 の競合相手数は 1，Thread2 の競合相手数は 3 であるため，Thread2 の実行するトランザクション Tx2 をアボートする．これによって，ストール状態にあった Thread1，Thread3，Thread4 は並列にリードアクセスが実行可能となる．なぜなら，これは `0x100` 番地に対する read after read のメモリアクセスであり，2.2.2 項に示した 3 つの競合パターンに該当しないためである．

ここで，Thread3 が `0x100` 番地へのリードアクセスではなく，ライトアクセスを実行しようとしていたと仮定する．同様にトランザクションの実行が進み，時刻 t5 において Thread3 が Tx3 の実行を再開する．このとき，同一アドレスへのリードアクセスを行うスレッドも同時に実行を再開するため，write after read のメモリアクセスとして競合が検出される．これにより，ストール状態から解放された後，最初にライトアクセスを実行した Thread3 以外のスレッドは，再びトランザクションをストールさせなければならない．このような場合，並列実行スレッド数の増加は見込めない．しかし，こうした場合においても，少なくとも 1 つ以上のスレッドはトランザクションの実行再開が可能であるため，提案する選択手法によって得られる効果は大きいと考えられる．

3.3.2 トランザクション開始時刻を比較

二つ目は，デッドロック検出時，最も遅く実行を開始したトランザクションをアボート対象とする手法である．開始時刻が早いトランザクションでは多くのメモリアクセスが行われている可能性が高く，このトランザクションを優先してコミットさせることで競合の頻発を防ぐことができると考えられる．また，開始時刻がより早いトランザクションの処理が常に優先されるため，スタベーションを防ぐことができる．

ここで，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がデッドロック状態に陥った様子を**図 6** に示す．時刻 t4 において，アボート対象を選択するため，3 つのトランザクション開始時刻 t1，t2，t3 を比較する．そして，Thread2 の実行する，最も遅く開始した Tx2 をアボートする．このように，より多くのメモリアクセスを行ったと考えられる Tx1，Tx3 を優先してコミットまで実行させることで，Thread2 はその後，他のスレッドと競合することなく再実行が可能となることが分かる．

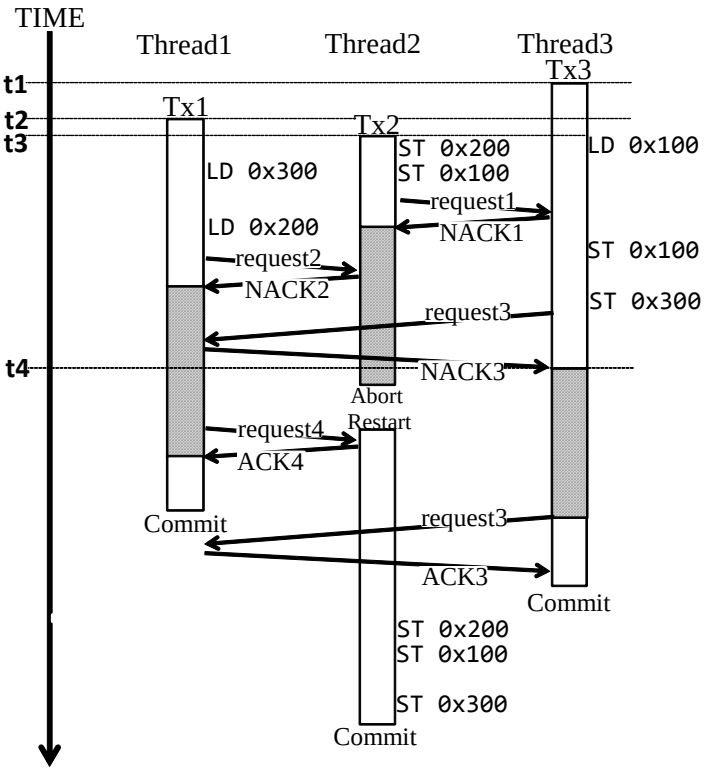

図 6: トランザクション開始時刻の比較による選択

4 依存関係情報の伝搬

本章では提案手法の具体的な実装方法について説明する．

4.1 拡張した LogTM の構成

アボート条件を厳格化するため，既存の LogTM を拡張し，以下の 3 つの記憶ユニットを各プロセッサコアに追加する．なお，コア数及び最大同時実行スレッド数は n であるとする．拡張した LogTM の構成図を**図 7** に示す．

stall_bits:

自身を直接的及び間接的にストールさせているスレッド番号を記憶する，n bit のビットマップ．

clear_bits:

依存関係の解消されたスレッド番号を記憶する，n bit のビットマップ．

conflict_bits:

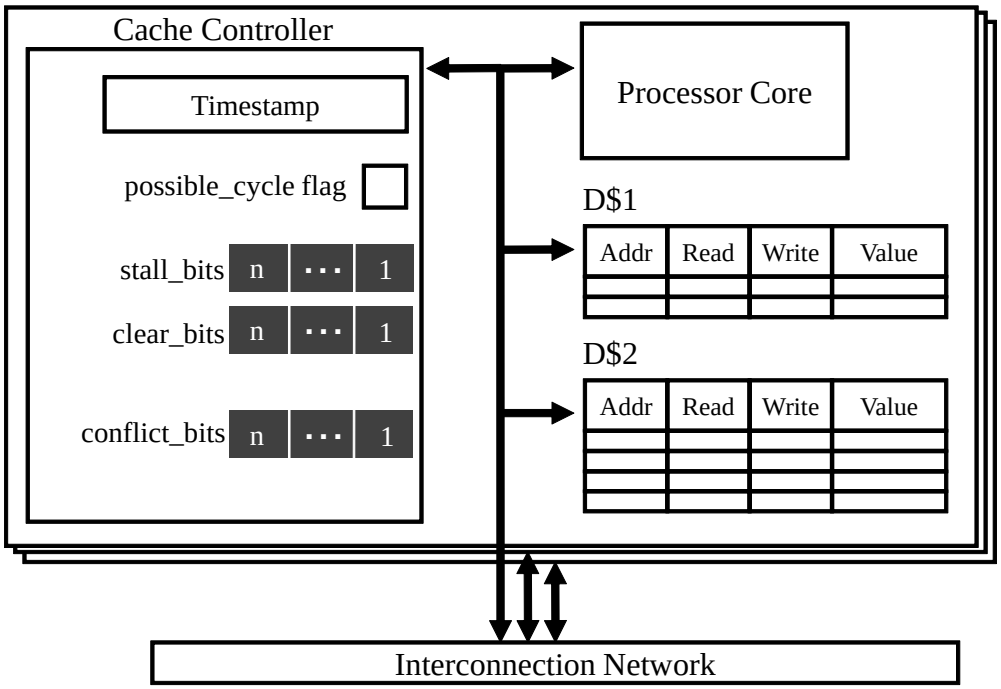

図 7: 拡張した LogTM の構成

自身が直接的にストールさせているスレッド番号を記憶する，n bit のビットマップ．なお，これは競合相手数を用いたアボート対象選択手法にのみ必要な機構である．

提案手法では，アボート条件をデッドロックが発生した場合のみとするため，依存関係の環の発生の有無を検査する必要がある．そのため，各スレッドが実行するトランザクションが，どのスレッドによってストールさせられているかを知る必要がある．そこで各スレッドは，自身の実行するトランザクションを直接的及び間接的にストールさせているスレッドの番号を，stall_bits として記憶する．さらに，競合時にこの stall_bits を NACK と併せて送信することで，依存関係情報を伝搬させる．そして NACK を受信した際に，このビット列に自身のスレッド番号に対応するビットがセットされていた場合，デッドロックの発生が検出される．

また，あるスレッドがトランザクションをコミット/アボートする際には，そのスレッドとの間に発生していた依存関係は解消されるため，各スレッドが保持するビット列に矛盾が生じることになる．そのため，これらのビットをクリアする必要がある．そこで，各スレッドは依存関係の解消されたスレッド番号を clear_bits として記憶する．そして，NACK 送信時にこの情報を付加し，NACK を受信したスレッドはこの clear_bits

を用いてコミット/アボート後に残存するスレッド間の依存関係情報を破棄する．

さらに，競合相手数を用いたアボート対象選択手法を適用する際には，各スレッドが，どのスレッドが実行するトンランザクションをストールさせているかを知る必要がある．そこで，自身が直接的にストールさせているスレッド番号を conflict_bits として記憶する．なお，これら stall_bits，clear_bits，conflict_bits の第 n ビットは，スレッド番号 n に対応している．

4.2 競合時の操作

3 つのスレッド Thread1，Thread2，Thread3 がそれぞれトランザクション Tx1，Tx2，Tx3 を投機的に実行する**図 8** を例に，追加したハードウェアに対する操作を説明する．各スレッドはそれぞれ stall_bits，clear_bits，conflict_bits を保持しており，各機構に記憶された値はアボート/コミット時にクリアされるが，stall_bits に限り，トランザクションの実行を再開する際にもクリアされる．

図 8(a) において，Thread1 が `ST 0x100` を，Thread3 が `LD 0x300` を実行し，その後に Thread2 が `ST 0x200` を実行済みである場合を考える．まず，Thread2 が `ST 0x300` を実行しようとすると，Thread3 は既に当該アドレスへのアクセスを行っているため競合を検出し，NACK を返信する (時刻 t1)．このとき，Thread2 に返信する NACK1 に stall_bits を付加することで，自身をストールさせているスレッド番号を伝搬させる．この例では，Thread3 の実行するトランザクション Tx3 は他のどのスレッドにもストールさせられていないため，stall_bits1 `000` を送信する．さらに，自身がストールさせたトランザクションを実行するスレッド番号を記憶するため，conflict_bits の該当するビットをセットする．Thread2 は NACK1 を受信すると (時刻 t2)，NACK1 とともに受信した stall_bits1 と自身の保持している stall_bits との論理和をとる．こうすることで，自身が実行するトランザクションを間接的にストールさせているスレッドの番号を知ることができる．さらに，NACK1 を送信したスレッドの番号に対応するビットをセットすること，自身のトランザクションを直接的にストールさせているスレッドの番号を記憶しておく．この論理和演算により Thread2 の保持する stall_bits `000` は `100` へと更新される．ここで，Thread2 は更新後の stall_bits に自身のスレッド番号に対応するビットがセットされているか否かを検査する．このとき，自身のスレッド番号に対応するビットはセットされていないことが確認されるため，デッドロックは検出されない．

その後，Thread1 が `0x200` 番地に対してアクセスする際，Thread2 と競合する様子

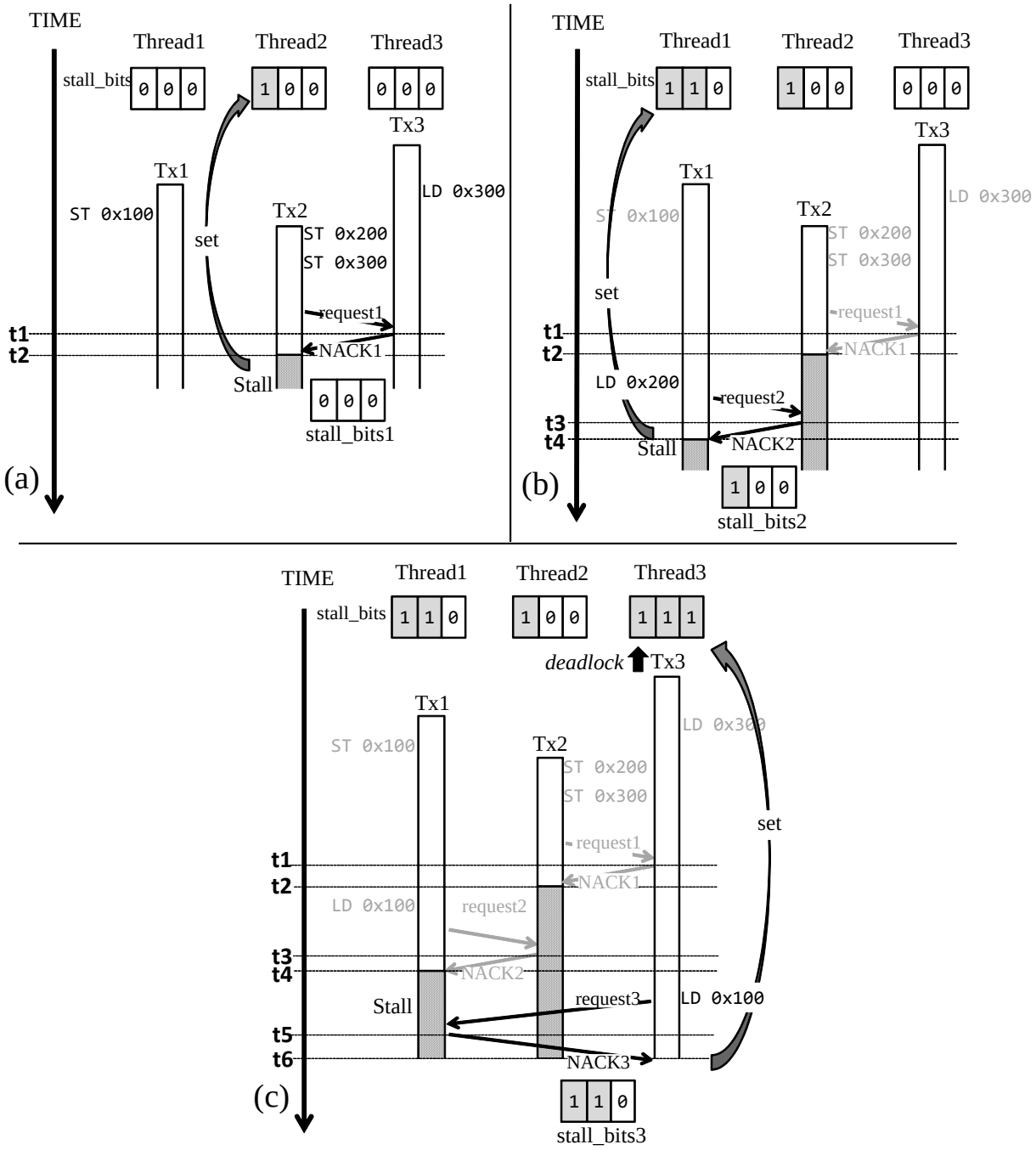

図 8: ビット列の伝搬によるデッドロックの検出

を図 8(b) に示す．Thread2 は Thread1 へ NACK2 を返信する際，Thread3 が NACK1 を返信した時と同様に，自身の保持する stall_bits2 100 を併せて送信する (時刻 t3)．また，Thread1 は NACK2 とともに受信したビット列と，自身の stall_bits の論理和をと

る．このビット演算により Thread1 の stall_bits 000 は 110 へと更新される (時刻 t4)．このとき，Thread1 は更新された自身の stall_bits から，Tx1 が Thread3 によって間接的にストールさせられていることを認識する．さらにその後，図 8(c) に示すように Thread3 が 100 番地に対してアクセスしようとする際に Thread1 と競合し，Thread1 から Thread3 へ NACK3 が送信される (時刻 t5)．NACK3 を受信した Thread3 は自身の stall_bits と，受信した stall_bits3 との論理和をとり，stall_bits 000 は 111 へと更新される (時刻 t6)．このとき，Thread3 は更新後の stall_bits に自身のスレッド番号に対応するビットがセットされていることを確認できるため，デッドロックの発生を検出する．

4.3 コミット/アボート時の操作

トランザクションをコミット/アボートする際には，競合の解決により依存関係が解消されるため，各スレッドが保持する stall_bits を正しく修正する必要がある．ここで，図 8(c) においてデッドロックを検出後，Thread3 が自身のトランザクションをアボートすることでデッドロックを解決する様子を**図 9**(a) に示す．このとき，Thread3 は自身の保持する stall_bits をクリアする．

その後，図 9(b) に示すように，Thread2 は ACK4 を受信することで Tx2 の実行を再開するため (時刻 t8)，自身の stall_bits をクリアする．しかし，Thread1 の保持する stall_bits は，Thread3 に対応するビットがセットされたままであるため，記憶している依存関係に矛盾が生じてしまう．そこで，依存関係の解消されたスレッド番号をビット列として表現したものを clear_bits として記憶し，NACK 送信時にこの情報を併せて送信する．NACK を受信したスレッドはこの clear_bits を用いて，依存関係の矛盾を解消する．ここで，Thread2 は ACK4 受信時 (時刻 t8) に自身の stall_bits をクリアする直前，このビット列を clear_bits 100 として保持しておく．トランザクションの実行が進み，Thread2 が request5 に対する NACK を返信する際，図 9(c) に示すように，stall_bits 000 に加えて clear_bits 100 を併せて送信する (時刻 t9)．NACK5 とともに 2 種類のビット列を受信した Thread1 は，自身の stall_bits と受信した stall_bits5 との論理和をとり，NACK 送信相手に対応するビットをセットする通常の操作に加え，受信した clear_bits5 に対応するビットをクリアする操作を行う．このビット操作により，Thread1 の stall_bits 110 は 010 へと更新され，既に依存関係の解消された Thread3 に対応するビットがクリアされる．一方コミットにより依存関係が解消された場合も，同様の手順でこのような間接依存関係を修正できる。

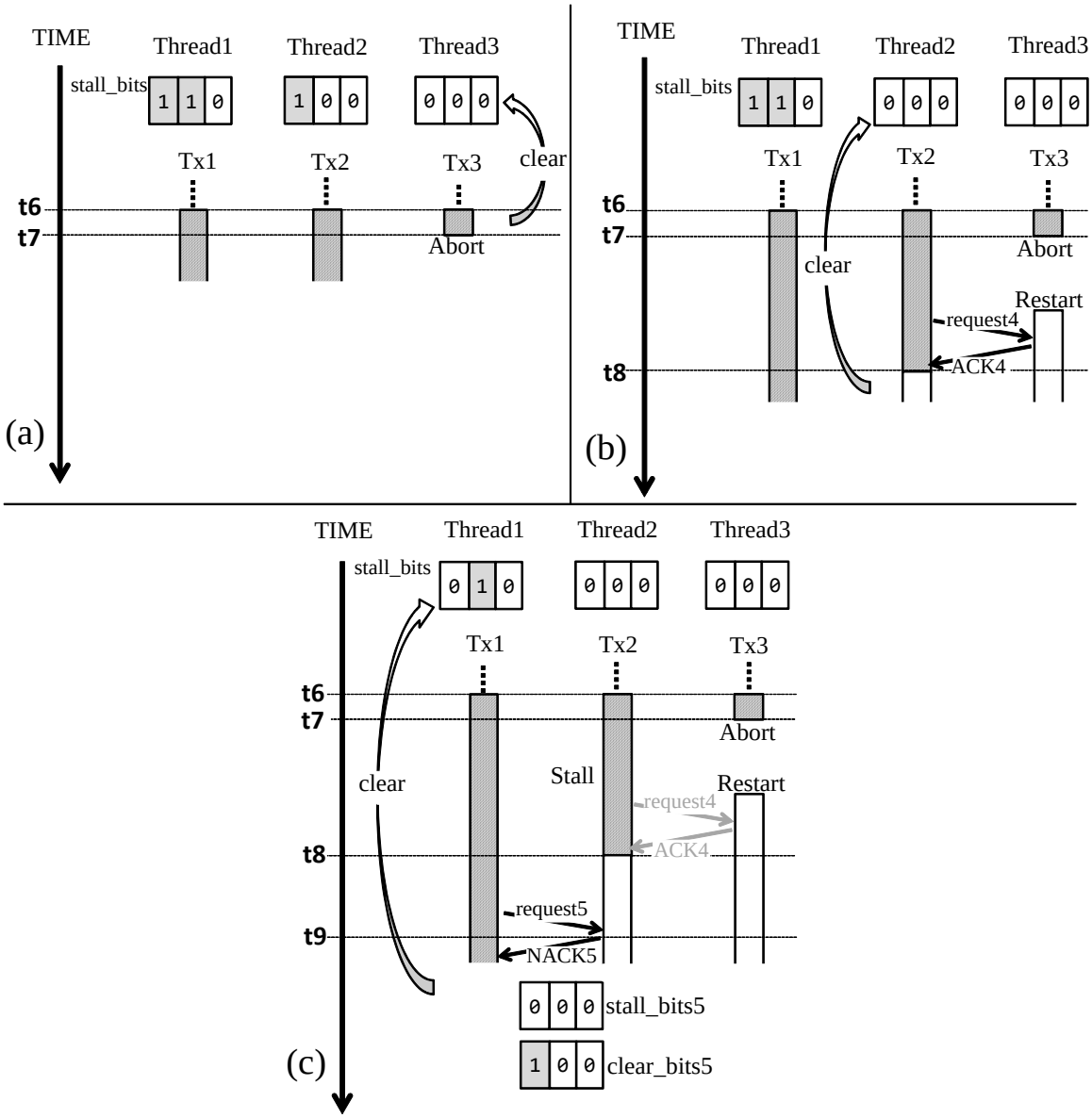

図 9: 依存関係の修正

4.4 アボート対象の選択

3.3 節で提案したアボート対象選択手法を適用するためには，デッドロックに関与しているトランザクションの情報を収集する必要がある．そのため，デッドロック検出後に，必要となる情報を NACK とともに伝搬させる．

情報を収集し，その情報に基づいてアボート対象を選択する様子を**図 10** に示す．まず，時刻 t1 において Thread1 が Thread2 と `0x100` 番地に対してアクセスしようとして

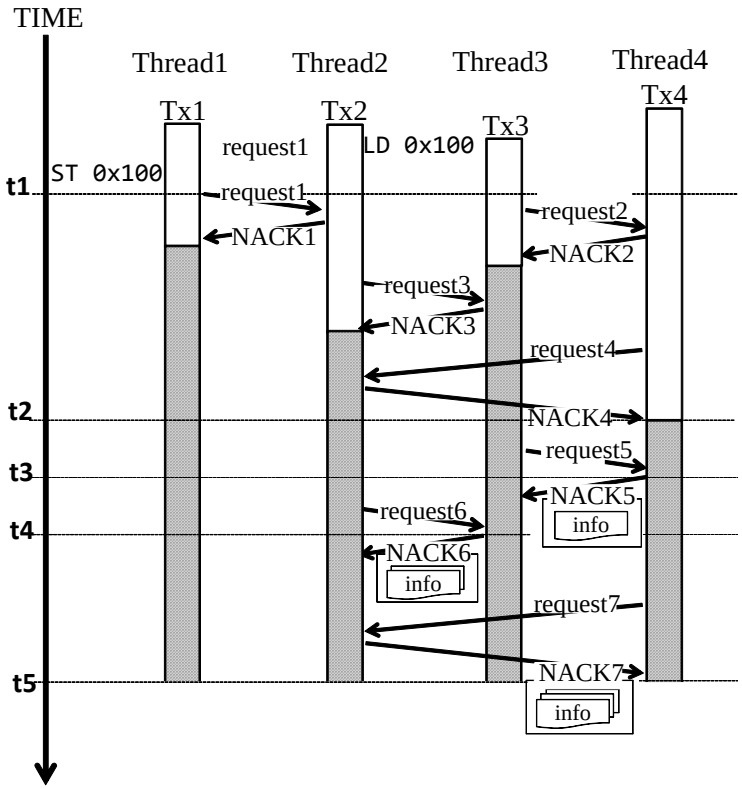

図 10: 情報収集によるアボート対象の選択

競合が発生した後，時刻 t2 において Thread2，Thread3，Thread4 の間でデッドロックが発生した場合を考える．デッドロックを検出した Thread4 は，Thread3 へ NACK5 を返信する際，競合相手数，または自身の実行するトランザクションの開始時刻を併せて送信する (時刻 t3)．なお，各スレッドは自身の conflict_bits を参照し，ビットがセットされた数をカウントすることで競合相手数を取得する．NACK5 を受信した Thread3 は，さらに自身の情報を NACK6 に付加し，これを Thread2 へ伝搬させる (時刻 t4)．また，NACK6 を受信した Thread2 も同様にして自身の持つ情報を Thread4 へ送信する．なお，情報を伝搬する際，Thread2、Thread3 及び Thread4 の間の循環依存関係は変化しないため，stall_bits を変更する必要は無い．このように，自身の持つ情報を NACK に併せて伝搬させることで，デッドロックを検出した Thread4 は，デッドロックに関与しているトランザクションを実行する，全てのスレッドの情報を収集することができる (時刻 t5)．そして，Thread4 は提案した以下 2 つの選択方法に基づきアボート対象を選択する．

スレッドの競合相手数: 自身がストールさせているトランザクションの数が最も多いトランザクションをアボート対象とする．このとき Thread4 は収集した情報から競合相手数を比較し，Tx1 をストールさせている Tx2 をアボート対象として選択する．
トランザクションの開始時刻: 最も遅く開始したトランザクションをアボート対象とする．このとき Thread4 は，最も遅く開始した Tx3 をアボート対象として選択する．選択したトランザクションをアボートさせるには，その旨を対象となるトランザクションを実行するスレッドに伝える必要がある．そこで，新たにアボートリクエストという通信メッセージを定義する．このリクエストはアボート対象として選択されたトランザクションを実行するスレッドへ送信され，リクエストを受信したスレッドはトランザクションをアボートさせる．

しかし，この情報収集は，特に 3 者以上のトランザクションがデッドロックに関与している場合，大きなオーバヘッドになる可能性がある．したがって，アボート対象の適切な選択によって大きな性能向上が見込めない限り，採用することは困難であると考えられる．本論文では，アボート対象の選択が性能に与える影響についても，評価，考察する．

5 評価

5.1 評価環境

これまで述べた拡張を LogTM に実装し，シミュレーションによる評価を行った．評価にはトランザクショナルメモリの研究で広く用いられている Simics[11] 3.0.31 と GEMS[12] 2.1.1 の組み合わせを用いた．Simics は機能シミュレーションを行うフルシステムシミュレータであり，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 Solaris10 とした．**表 1** に詳細なシミュレーションパラメータを示す．

5.2 評価結果

評価対象のプログラムとしては，GEMS 付属の microbench から btree，contention，deque，prioqueue，SPLASH-2[13] から cholesky，radiosity の計 6 種のベンチマークプログラムを用い，それぞれのプログラムを 8，16 及び 31 スレッドで実行した．それぞれの入力を**表 2** に, 評価結果を**表 3** および**図 11** に示す．図 11 中の凡例はサイクル数の内訳を示しており，それぞれは以下の通りである．

stall: ストールに要したサイクル数

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
IPC	non-memoryIPC=1
D1 cache	32 KBytes
ways	4 ways
latency	1 cycle
D2 cache	8 MBytes
ways	8 ways
latency	20 cycles
Memory	4 GBytes
latency	450 cycles
Interconnect network latency	14 cycles

表 2: ベンチマークプログラムの入力パラメータ

GEMS	
contention	config 1
prioque	8192ops
btree	priv-alloc-20pct
deque	1024ops 32bkoff
SPLASH2	
cholesky	tk14.0
radiosity	-p 31

barrier: バリア同期に要したサイクル数
backoff: アボート後に実行開始までランダム時間待つサイクル数
aborting: アボート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

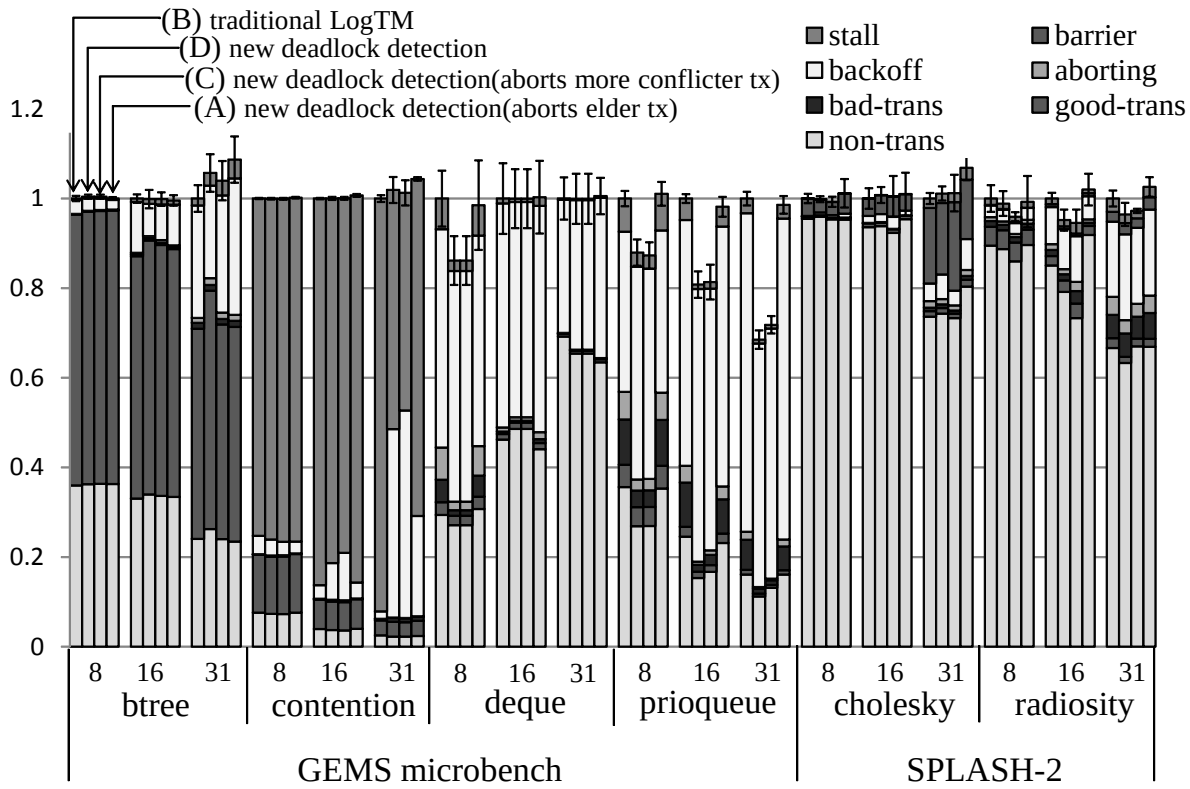

図 11: 各プログラムにおけるサイクル数比

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

LogTM では，アボート直後にトランザクションを再開した場合，同じ競合が再度発生することを防ぐため，アボート後にランダムサイクル待機する backoff 機能を備えている．なお，この待機期間はアボートが発生するたびに指数関数的に増大するよう設定されている．

図中では，各ベンチマークプログラムと前述の実行スレッド数との組合せによる結果が，各 4 本のグラフで表されている．4 本は左から順に，それぞれ

(B) 既存モデル（ベースライン）

(D) デッドロックを検出したスレッドが自身のトランザクションをアボートする提案モデル

(C) 競合数の多いスレッドの Tx をアボートする提案モデル

(A) 実行開始時刻の遅い Tx をアボートする提案モデル

表 3: 各ベンチマークにおける削減サイクル数

		(D)		(A)		(C)	
bench	/thrs	最大	平均	最大	平均	最大	平均
GEMS	/8	13.9%	6.6%	1.5%	0.7%	13.9%	6.8%
	/16	19.2%	5.2%	1.8%	0.3%	18.6%	5.0%
	/31	31.5%	7.3%	1.4%	-3.0%	28.2%	6.8%
SPLASH-2	/8	1.1%	0.7%	0.7%	-0.2%	4.1%	2.4%
	/16	4.8%	2.1%	-1.0%	-1.5%	5.4%	2.5%
	/31	3.5%	1.2%	-2.6%	-4.7%	2.8%	0.8%

の実行サイクル数の平均を表しており，既存モデル (B) の実行サイクル数を 1 として正規化している．なお，提案モデル (C)，(A) のサイクル数は情報収集にかかるコストを含んでいない．

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す．

シミュレーションによる評価の結果，デッドロックを検出したスレッドが自身の実行するトランザクションをアボートするモデル (D) では，ほとんどの場合において性能が向上した．一方，競合者相手数を比較するモデル (C) は既存モデル (B) に対しては多くの場合速度向上しているものの，(D) に対しては目立った速度向上が得られておらず，情報収集のためのコストも考慮すると，(D) が優れていると言える．また，開始時刻の遅いトランザクションをアボートする提案モデル (A) では，既存モデル (B) に対して性能悪化する場合も見られた．

次節では，各ベンチマーク別に詳細な検証を行う．

5.3 考察

GEMS microbench

まず GEMS microbench の結果を見ると，提案モデル (D) において deque，prioqueue では aborting，bad-trans，stall が削減されており，アボートの発生が抑制されたことが分かる．結果として，31 スレッドでは最大 31.5%，平均 7.3%の実行サイクル数が削

表 4: btree31 スレッドにおけるトランザクション別アボート回数

btree/31	命令数	(B)	(D)
Tx1	多い	2199.6	2305.5
Tx2	少ない	1037.3	85.3

表 5: contention31 スレッドにおける平均アボート回数及び最大アボート繰り返し回数

contention/31	(B)	(D)
平均アボート回数	549.2	361.1
平均最大アボート繰り返し回数	13.8	21.0

減された.

しかし，btree を 31 スレッドで実行した場合にはどのモデルにおいても性能が低下していた．btree で実行されるトランザクションは，命令数の多いものと少ないものの二つに大別される．ここで，(B) 及び (D) において，btree を 31 スレッドで実行したときの，各トランザクションで発生した平均アボート回数を表 4 に示す．この結果から，提案モデル (D) では，Tx2 がアボートされた回数が (B) に比べて大きく削減されていることが分かる．それにも関わらず，(D) の実行サイクル数は 31 スレッドで実行した場合，既存モデル (B) に比べて最大 5.7%増加していた．これは，アボート対象として選択されるトランザクションが命令数の多い Tx1 に偏ったために，アボート時のログの書き戻し量が増加し，ロールバックによるオーバヘッドが増大したためであると考えられる.

また，contention では stall サイクルが大きく削減されたが，同時に backoff サイクルが増加し，既存モデル (B) に対する性能向上は得られなかった．提案モデル (D) における contention の平均アボート回数及び平均最大アボート繰り返し回数を**表 5** に示す．この結果から，提案モデル (D) では平均アボート回数が減少する一方で，同一トランザクションがアボートを繰り返す回数が増加していることが分かる．このようにトランザクションが連続してアボートされることで，backoff が大きく増加してしまったと考えられる.

一方提案モデル (C) では，既存モデル (B) に対して 31 スレッドで最大 28.2%，平均 6.8%サイクル数を削減し，並列実行スレッドの増加によりサイクル数が削減されることを，多くのプログラムで確認した．しかし，prioqueue を 16 及び 31 スレッドで実

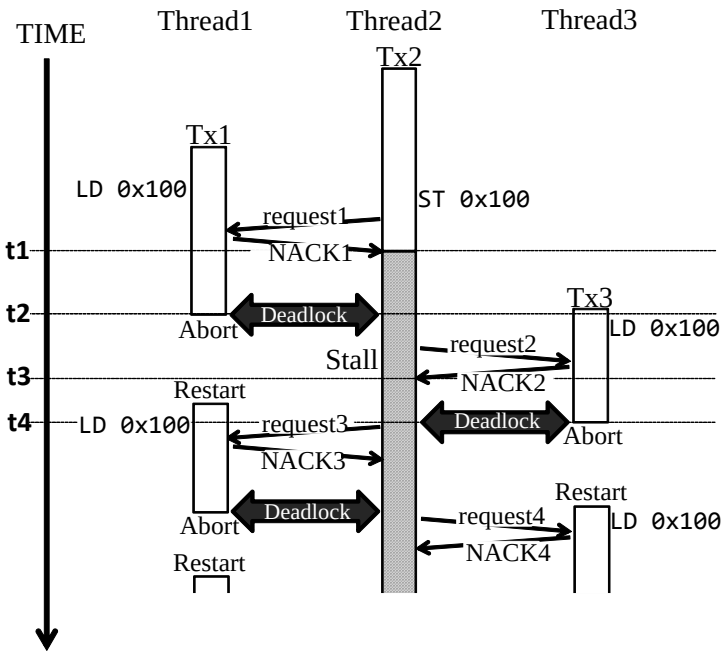

図 12: starving writer の発生

行した場合には，提案モデル (D) に対する性能向上は得られなかった．これは，競合相手数のより多いトランザクションをアボートさせることで並列実行スレッドが増加した一方で，競合しやすいトランザクションがアボートされたため，再実行後も再度他のトランザクションとの間で競合が発生したことが原因であると考えられる．したがって，条件次第では競合相手数の多いトランザクションを優先してコミットさせる手法を適用する必要があると考えられる．

また，提案モデル (A) では提案モデル (D) に比べてアボートが繰り返し発生したために，aborting，bad-trans が大きく増加していた．これは，starving witer と呼ばれるトランザクションが発生する競合パターンに陥ったためであると考えられる．この競合パターンは，あるメモリアドレスへのストアアクセスが複数のロードアクセスにより妨げられ続けることにより発生する．starving witer が引き起こされる様子を**図 12** に示す．この例では 3 つのスレッド Thread1，Thread2，Thread3 がそれぞれトランザクション Tx1，Tx2，Tx3 を投機的に実行している．まず，Thread2 が最も早くトランザクションの実行を開始し，`ST 0x100` を実行しようとする場合を考える．このとき，Thread1 は競合を検知するため，NACK を返信し，Thread2 は Tx2 をストールさせる (時刻 t1)．この後，Thread1，Thread2 の間でデッドロックが発生するような

メモリアクセスが実行されたと仮定する．簡単のため，図ではこれを Deadlock と示す (時刻 t2)．モデル (A) ではトランザクション開始時刻を比較することでアボート対象を選択するため，Thread1 の実行する Tx1 をアボート対象として選択する．このとき，Thread1 がアボートによるログの書き戻し処理を完了するまで Thread2 は `0x100` へライトアクセスすることができない．一方で，Thread3 は `LD 0x100` に対してリードアクセスすることができる．Thread1 のアボート処理終了後，Thread2 は `ST 0x100` の実行を試みるが，`0x100` 番地へのアクセスは既に Thread3 によって行われているため，トランザクションをストールから復帰させることができない (時刻 t3)．次に Thread2 と Thread3 の間でデッドデッドロックが発生した際 (時刻 t4) も，提案手法により Tx3 がアボート対象として選択されてしまうためこの動作が繰り返される．その後も，ライトアクセスを行おうとする Thread2 は Tx2 をストールさせたまま，リードアクセスを行うスレッドのトランザクションをアボートさせ続ける状態となる．このように，(A) を適用した場合には，リードアクセスを行うトランザクションがアボート対象として選択されたために，アボート回数が増加したと考えられる．したがって，アボート対象の選択基準にはライト/リードのどちらのアクセスによる競合かを考慮する必要があると考えられる．

SPLASH-2

まず，提案モデル (D) において、radiosity では aborting、bad-trans、non-trans が削減されたことから、アボートが抑制されたことが分かる。一方、cholesky では既存モデル (B) に対する性能向上は得られなかった。これは、プログラム実行中に発生したデッドロックの多数が 2 者間におけるものであり、possible_cycle flag を用いた競合解決に近い動作となったためであると考えられる。

また，競合相手数を比較するモデル (C) に関して，cholesky は競合の少ないプログラムであるため競合相手数に差が生まれにくく，目立った効果は得られなかった．一方 radiosity では，ひとつのライトアクセス済スレッドに対して，リードアクセス許可を待つ多数のスレッドが存在する状況が見られ，これらのスレッドを並列実行させることでサイクル数の削減を達成できた．

一方，トランザクション開始時刻を比較するモデル (A) では，ほとんどの場合において性能が悪化していた．これは，cholesky，radiosity の両方において，GEMS 同様 starving writer の発生によるものであることが分かった．これらのプログラムは規模の小さいトランザクションを多く含んでいるが，starving writer の発生により，本来であればすぐに実行を完了するこれらのトランザクションが，コミットまでに長いサ

イクルを要するようになっていた．さらに non-trans も増加していることから，このトランザクションの処理の遅れが，トランザクション外におけるキャッシュアクセスとの false sharing を増大させていると考えられる．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，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競合として検出されてしまう，偽の共有状態のことである．したがって，今後これらを詳細に調査し，対策を検討していく必要がある．

6 おわりに

本論文では，既存のハードウェア・トランザクショナル・メモリである LogTM を拡張し，トランザクションをアボートする条件をデッドロック検出時のみとすることで，アボートの発生を抑制する手法を提案した．また，スレッドの競合相手数及びトランザクションの実行開始時刻に着目し，デッドロックに関係するトランザクションの中から適切なアボート対象を選択する手法を提案した．提案手法の有効性を確認するため，GEMS microbench および SPLASH-2 ベンチマークプログラムを用いて評価した結果，possible_cycle flag を用いた既存モデルに比べて最大 31.5%，平均 7.3%の実行サイクルが削減されることを確認した．提案手法の効果が得られた要因として，アボートを抑制したことによる，ロールバック時の書き戻しコストの減少や再実行サイクル数の削減が挙げられる．しかし，デッドロック検出時にアボート対象を選択した結果，既存の LogTM よりも性能が悪化してしまう場合も見られた．これには同一トランザクションのアボートの繰り返しによる Backoff サイクルの増加や，starving writer と呼ばれるトランザクションが発生する競合パターンが影響していた．

今後の課題として，サイクル削減率が最も悪化した，実行開始時刻の比較によりアボート対象を選択する提案モデルの改良が挙げられる．具体的には，アボート対象の選択基準にライト/リードのどちらのアクセスによる競合かを考慮する手法を考案していく．スレッドの競合相手数の比較によりアボート対象を選択する提案モデルに関しても，競合相手数の多いスレッドを優先して実行すべき条件を洗い出し，あらゆる競合パターンに対応できる手法を考案していく．

また，アボート対象選択部分は現在実装途中であるため，アボート対象選択のための情報収集にかかるコストを見積もることができなかった．したがって，このコストを含めた実行サイクル数を求める必要がある．それに加え，情報収集にかかるコストはベンチマークプログラムにおける競合のしやすさや，並列実行スレッド数に大きく依存すると考えられるため，効率的にアボート対象を選択する方法について今後検討

していきたい.

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，斎藤彰一准教授，松井俊浩准教授に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室および齋藤研究室，松井研究室の方々に深く感謝致します．特に、浅井宏樹氏，江藤正通氏には研究を進めるにあたって多大な助言を頂きました．ここ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. of 20th Annual Int'l Symp. on Computer Architecture*, pp. 289–300 (1993).

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

[3] Shavit, N. and Touitou, D.: Software Transactional Memory, pp. 204–213 (1995).

[4] Sweazey, P. and Smith, A. J.: A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus, *Proc. of 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 Transactions 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 of 10th Symp. on Architectural Support for Programming Languages and Operating Systems*, pp. 5–17 (2002).

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

[8] 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).

[9] 伊藤悠二, 塩谷亮太, 五島正裕, 坂井修一: 最適なロールバック・ポイントを選択するトランザクショナル・メモリ, 先進的計算基盤システムシンポジウム SACSIS2011 論文集, pp. 324–331 (2011).

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

[11] 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).

[12] 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).

[13] Woo, S. C. et al.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc of 22nd Int'l. Symp. on Computer Architecture (ISCA'95)*, pp. 24–36 (1995).

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