修士論文

競合検出単位の細粒度化によるLogTMの高速化

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 22 年度入学　22413502 番

浅井 宏樹

平成 24 年 2 月 3 日

競合検出単位の細粒度化による LogTM の高速化

浅井 宏樹

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制御にロックが広く利用されている．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競合として検出する．

この TM をハードウェアで実現したもものひとつである LogTM は，各キャッシュラインに read 及び write ビットを付加しており，これらのビットを使用してトランザクション内で発生したリード及びライトアクセスの情報を記憶することで，キャッシュライン単位で競合を検出している．競合が発生した場合，競合相手のトランザクションが終了するまで処理を一時的に中断，またはそれまでの実行結果を破棄することで，競合を解決する．

しかし，LogTM では個々の変数に対するアクセスの競合を検出することはできない．そのため，他のトランザクションでアクセスされた変数とは異なる変数にアクセスしたとしても，それらの変数が同一キャッシュライン上に配置されているならば，競合として判定されてしまうという問題が発生する．

そこで本研究では，競合の検出単位を細粒度化する 2 つの手法を提案する．これらの手法により，本来ならば検出する必要のない無駄な競合の発生を防止する．1 つ目は，このような競合が発生する可能性のある変数を自動的に異なるキャッシュラインに分散配置させる手法であり， 2 つ目は，各変数へのアクセス情報を記憶する表をハードウェア上に保持することで，変数単位で競合を検査可能とする手法である．さらにこれら 2 つの手法を組み合わせた手法も提案する．

提案した手法の有効性を検証するために，既存の LogTM に提案手法のモデルをそれぞれ実装し，STAMP 及び GEMS 付属のマイクロ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，1 つ目の提案手法では最大 83.9%の実行サイクル数が削減できたが，平均では 7.0%の悪化となった．一方で，2 つ目の提案手法では，最大 82.9% ，平均 28.0%のサイクル数の削減を確認した．

競合検出単位の細粒度化による LogTM の高速化

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，SIMD や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，これらの手法だけではプロセッサの性能向上は頭打ちになりつつある．また，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，プロセッサの動作周波数の向上は困難になってきている．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．例えば，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最適なロックの粒度を設定しなければ並列性の向上が難しい．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 [1] が提案されている．

トランザクショナル・メモリをハードウェアで実現したもののひとつである LogTM[2] は，クリティカルセクションを含む一連の命令列であるトランザクションを投機的に実行する．また，投機実行されるトランザクションのアトミシティを保つために，あるアドレスが複数のトランザクションによりアクセスされたかどうかを検査し，競合の発生を検出する．この LogTM には，各キャッシュラインに対して read 及び write ビットという，トランザクション内で発生したリード及びライトアクセスの有無を保持するフィールドが追加されている．そして，キャッシュの一貫性を保つための要求を受け取ったときにこれらのビットを参照することで，メモリアクセスの競合をキャッシュライン単位で判定する．このとき，競合が発生したと判定されたトランザクションを実行するスレッドは，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，一種のデッドロック状態に陥る可能性がある．このような状況では，片方のトランザクション内でそれまでに完了された実行結果を全て破棄するアボート操作が行われる．アボートされたトランザクションでは，そのトランザクションが開始した時点のメモリ及びレジスタ状態が復元され，再実行される．

しかし，既存の LogTM はキャッシュライン単位で競合を検査しているため，他のトランザクション内でアクセスされた変数とは異なる変数にアクセスしたとしても，それらの変数が同一キャッシュライン上に配置されているならば，競合として判定されてしまう．このため，本来ならトランザクションの実行を停止する必要がない状況でも停止してしまうという問題がある．さらに，プログラマにより並行性制御の対象から除外された変数に対するアクセスでも競合として判定される可能性があり，そのような変数はトランザクション内部でアクセスされることが保証されないため，トランザクション以外の処理にも影響を与えてしまう可能性がある．

そこで，本研究では競合を検出する単位を細粒度化することで，本来ならば競合として検出する必要のない無駄な競合の発生を防止する 2 つの手法を提案する．1 つ目は，無駄な競合が発生する可能性のある変数を自動的に異なるキャッシュラインに分散配置させる手法であり，2 つ目は，各変数へのアクセス情報を記憶する表をハードウェア上に保持することで，変数単位で競合を検査可能とする手法である．前者は既存のハードウェアを拡張する必要なく無駄な競合の発生を防止できるが，データの局所性が低減しキャッシュミスが増大する恐れがある．一方で，後者は既存のバイナリをそのまま利用できるという利点を持つが，既存手法に比べて必要なハードウェアコストが増大するという欠点がある．

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 の欠点及びその解決方法を説明し，4 章及び 5 章では本研究で提案する 2 つの手法とそれらを融合した手法について説明する．6 章でそれらを評価し，7 章で関連研究について説明する．最後の 8 章において結論を述べる．

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・メモリと，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，共有メモリ型の並列プログラム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制御する必要がある．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，最適なロックの粒度を設定することは困難である．例えば，粗粒度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**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，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．これにより，複数のトランザクションによる，同一アドレスへのアクセスが確認された場合，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**という．競合が発生したと判定された場合，片方のトランザクションの実行を停止し，それまでの結果を全て破棄する．これを**アボート**という．トランザクションをアボートしたスレッドはトランザクション開始時点から再実行する．一方でトランザクションが終了するま

でに競合が発生しなかった場合，トランザクション内で更新されたすべ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，アボート及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．

ここで，TM で行われる競合の検出，コミット及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)[3] と呼ばれる．STM では，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．

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 Memory (LogTM)** について述べる．

2.2.1 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，LogTM では各キャッシュラインに対して **read ビット**及び **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リードアクセス及びライト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に，LogTM ではキャッシュの一貫性を保持するプロトコルであるディレクトリベース [4] の Illinois プロトコル [5] を拡張している．一貫性プロトコルでは，スレッドがあるメモリアドレスにアクセスする場合，キャッシュラ

インの状態を変更させるリクエストが他の各スレッドに送信される．拡張した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 及び 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:
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してリードアクセスすることが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:
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してライトアクセスすることがプロトコルにより検出された場合である．このとき，トランザクションがコミットされる前に，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たことで，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:
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してライトアクセスすることがプロトコルにより検出された場合である．このとき，トランザクションがコミットされる前に，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たことで，write after read 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，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．NACK を受信したスレッドは自身のアクセスで競合が発生したことを知るが，すぐにはアボートせず，競合を検出したスレッドの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．このとき，ストールされたトランザクションは競合したアドレスに対する

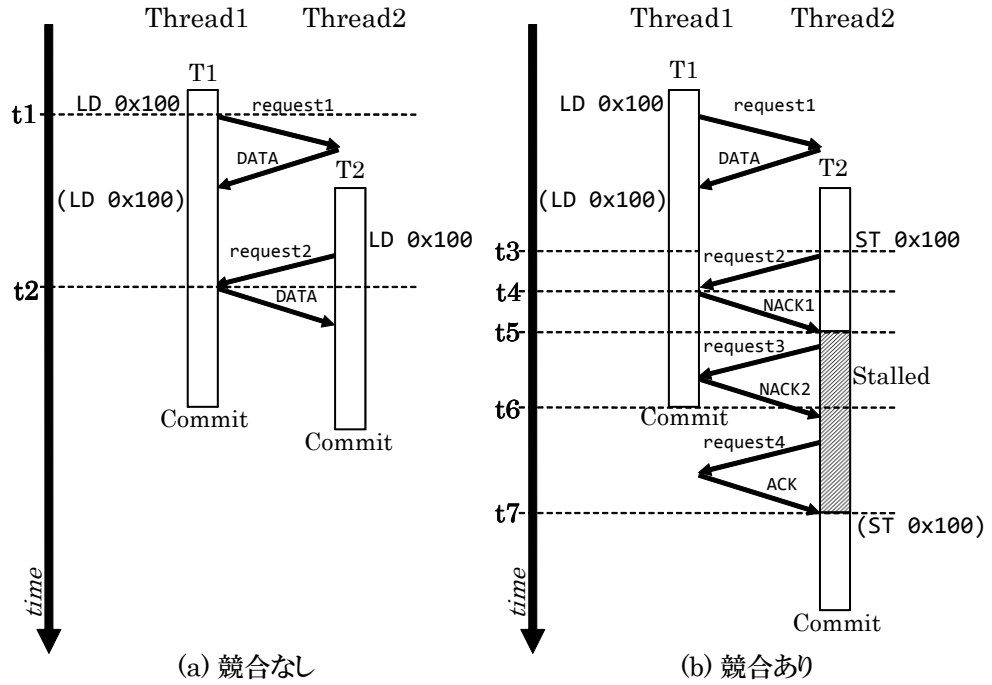

図 1: LogTM における競合の検出

リクエストを送信しつづけることで，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．例えば，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，並列に実行している 2 つのトランザクションの間で発生する競合を検出する動作モデルを**図 1** に示す．図 1 中の *time* は下に向かって時間が進むことを示しており，Thread1 と Thread2 はそれぞれスレッドを示し，それらのスレッドはそれぞれトランザクション T1 と T2 を投機的に実行しているとする．また図 1 中にあるメモリアドレスは全て共有メモリ空間上のメモリ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 1(a) を用いて説明する．図 1(a) では， Thread1 が時刻 t1 で Thread2 より先に 0x100 番地に対するロード命令を実行している．このとき，Thread1 は命令を実行する前に 0x100 番地に対するアクセスリクエストを Thread2 へ送信する．この時点では Thread2 は 0x100 番地へのアクセスは行っていないため，Thread1 はロード命令を実行することができる．その後 Thread2 が 0x100 番地に対してロード命令を実行しようとすると，Thread2 は命令を実行する前に 0x100 番地に対するリクエストを

送信する．すると，時刻 t2 で Thread1 はリクエストを受信することにより，Thread2 が 0x100 番地にアクセスしようとしていることを知るが，先ほど説明した 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，Thread1 は Thread2 に対して 0x100 番地のデータを返信する．すると，データを受信した Thread2 は命令を実行することができる．

次に，図 1(b) を用いて競合が発生する場合について説明する．図 1(a) の場合と同様に Thread1 は 0x100 番地に対するロード命令を実行する．その後，時刻 t3 で Thread2 が 0x100 番地に対してストア命令を実行しようとする．このとき，Thread2 は命令を実行する前に Thread1 へ 0x100 番地に対するリクエストを送信する．しかし，このアクセスは前述の競合パターンのうち write after read の条件を満たすため，競合として検出される．このため，Thread2 からのリクエストを受信した Thread1 は，競合したことを通知するために Thread2 へ NACK を返信する (時刻 t4)．すると，時刻 t5 で Thread2 は NACK を受信し，ストールする． Thread2 はトランザクション T2 をストールしている間もリクエストを再送し続け，0x100 番地へのアクセス許可を待つ．トランザクション T2 の実行が進み，時刻 t6 で Thread1 がコミットすると， Thread2 は再送したリクエストに対する ACK を受信することで， 0x100 番地へアクセスできるようになったことを知る．そこで，時刻 t7 で Thread2 はトランザクション T2 をストール状態から復帰させ，実行を再開する．

しかし**図 2**(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，一種の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 Thread1 と Thread2 がそれぞれトランザクション T1 及び T2 を投機的に実行している．まず，Thread1 が T1 の実行を開始した後に Thread2 が T2 の実行を開始しており，Thread1 が `ST 0x100` を実行し，その後に Thread2 が `ST 0x200` を実行済みであるとする．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 0x200 に対するアクセスリクエストを送信する (時刻 t1)．リクエストを受信した Thread2 は競合したことを検知するため NACK を返信する (時刻 t2)．NACK を受信した Thread1 は T2 が終了するまでストールする (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 に対して定期的にリクエストを送信している．この後，Thread2 が `LD 0x100` の実行を試みる (時刻 t4) と，Thread1 は Thread2 と競合し，T2 をストールさせる (時刻 t5)．このように，2 つのスレッドで実行中のトランザクションはお互いに相手のトランザクションが終了するまでストールする場合，これ以上処理を進めることができなくなる．

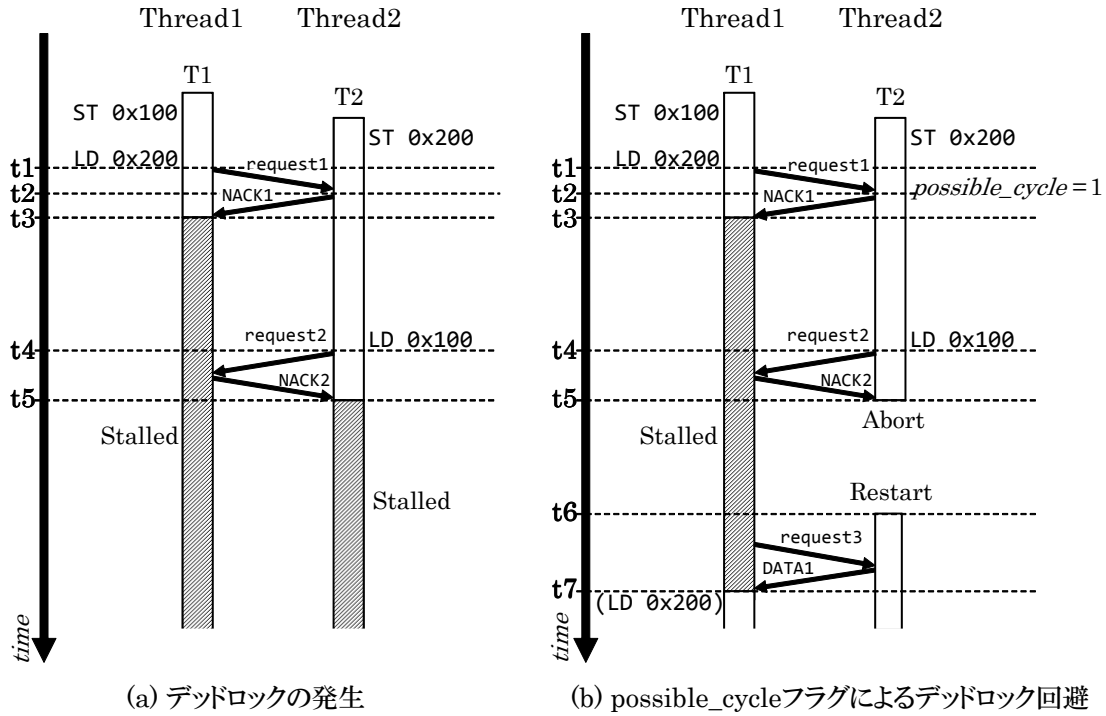

図 2: LogTM におけるデッドロック状態の解決

このようなデッドロック状態を回避するために， LogTM では 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い開始時刻を持つ場合，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，デッドロックを possible_cycle フラグにより回避する例を図 2(b) に示す．Thread2 は自身より早くトランザクションを開始した Thread1 に NACK を返信する際に possible_cycle フラグをセットする (時刻 t2)．そして，possible_cycle フラグがセットされている Thread2 は，自身よりも早くトランザクションを開始した Thread1 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 (時刻 t5)．このように，開始時刻の遅いトランザクションを実行しているスレッドがアボートの対象として選択される．これにより，早く開始したトランザクションを優先してコミットさせることができるため，すべ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されるようになる．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．アボート操作を行った Thread2 はトランザクション開始時点まで戻り再実行す

る (時刻 t6)．また，Thread2 がアボート操作を行ったことにより，Thread1 は 0x200 番地にアクセスできるようになるため，Thread1 のストール状態が解消される (時刻 t7)．

2.2.2 データのバージョン管理

LogTM 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，ライトアクセスにより更新したデータと，更新される前の古いデータを保持し管理する必要がある．このようなデータの管理をバージョン管理という．このバージョン管理は，LogTM ではスレッドごとに割り当てられた**ログ領域**と呼ばれる仮想メモリ領域に対して古いデータを退避することで実現されている．ログ領域には，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を含むキャッシュラインのデータ領域と，そのラインのアドレスとが退避される．これは， 2.2.1 項で説明したように，LogTM がキャッシュライン単位で競合を検査しているためである．一方で，ストア命令の結果である更新したデータはメモリ上に記憶される．なお，トランザクションの再実行に必要となるメモリ状態はトランザクション開始時点のもののみである．そのため，トランザクション内で同じまたは同一キャッシュライン上に存在するアドレスに対して複数回ストア命令が実行されたとしても，ログ領域には最初にストア命令を実行した時点のデータのみ退避しておけばよい．

ここで，LogTM におけるバージョン管理の動作について**図 3** を用いて説明する．図 3 ではあるスレッドがあるトランザクションを投機的に実行する様子を (a) から (d) に示しており，Thread，Cache 及び Thread's Log はそれぞれトランザクションを実行するスレッド，スレッドからアクセスされるローカルキャッシュ，及びスレッドに割り当てられたログ領域を表す．キャッシュにはキャッシュタグ，データ及び read/write ビットフィールドが存在しており，このキャッシュラインには int 型のデータを 4 つ保持できるとする．また，主記憶には**図 4** のように配列 `a[0-3]` が 0x100 番地以下に配置されているとする．便宜上，主記憶のメモリをキャッシュライン単位で表すとする．

まず，スレッドがトランザクションを開始したときの様子を図 3(a) に示す．このとき，Cache には 0x100 番地から始まるキャッシュラインが保持されているとする．その後，トランザクションの実行が進行し，`ST a[0],26` の実行を試みる場合を考える (図 3(b))．このとき，ストア命令を実行する前に，ストアにより更新される値が存在しているキャッシュラインのデータ領域及びそのラインのアドレスである 0x100 番地がログ領域に退避される (図 3(b)①)．その後，ストアの結果である値 26 がキャッシュに上書きされる (図 3(b)②)．

ここで，投機的実行が成功した場合，トランザクションはコミットされる (図 3(c))．

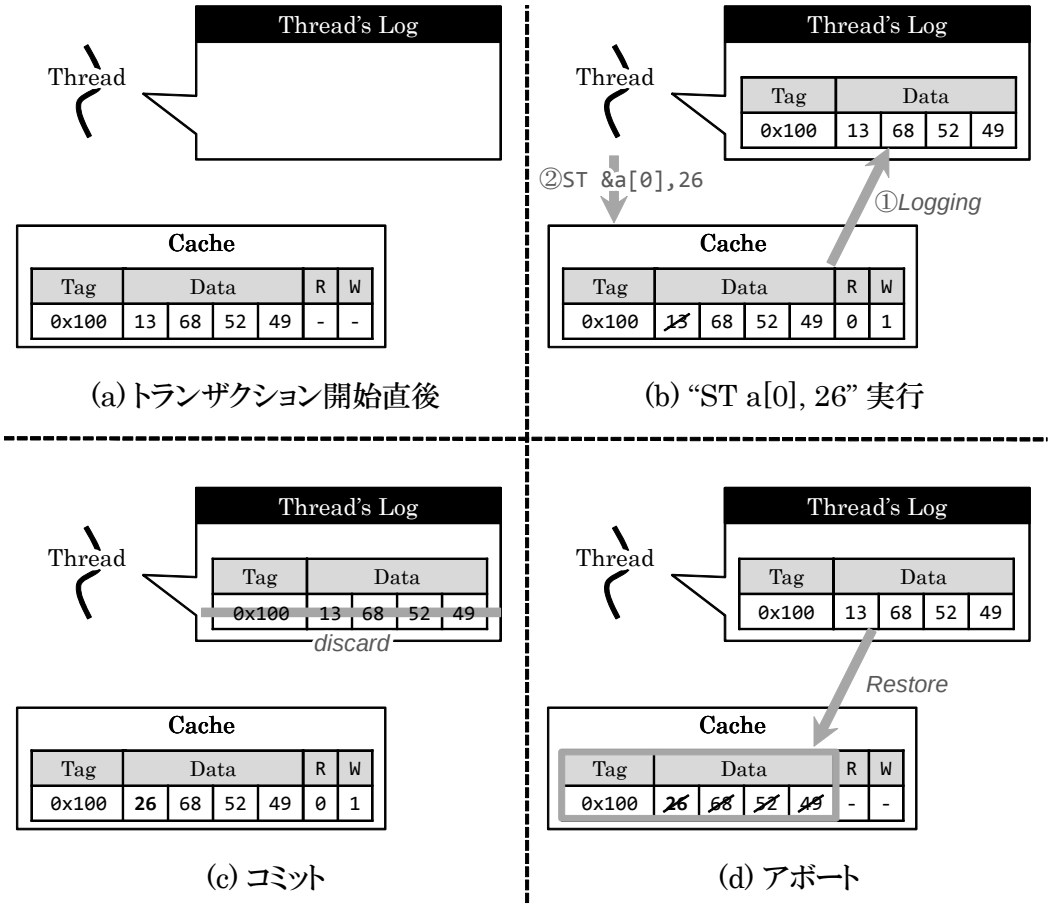

図 3: LogTM におけるデータのバージョン管理

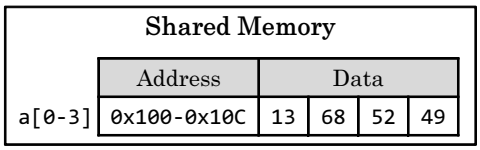

図 4: 想定するメモリ状態

このとき，値 26 は既にキャッシュに保持されているため，トランザクション内における実行結果は既にメモリ上に反映されている．したがって，ログ領域の内容を破棄する操作を行う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，トランザクションはアボートされる．アボート時の操作では，図 3(d) のように，ログ領域に退避されたキャッシュライン上の全てのデータが元のメモリアドレスに書き戻される．これにより，トランザクション開始

時点のメモリ状態を復元することができる．

また，アボート後にトランザクションを再実行するためには，メモリ状態と同様にレジスタ状態をトランザクションの開始時点まで戻す必要がある．したがって，LogTM 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，その状態をログ領域に保持しておく．そして，アボート時にログ領域を参照することでトランザクションの開始時点の状態を復元する．

LogTM ではこのようなアボート操作をアボートハンドラが担当している．アボートハンドラはアボート時に呼び出され，ログ領域を走査し，退避された情報を基にメモリ状態とレジスタ状態を復元する．図 3(d) の例の場合，まずアボートハンドラはログ領域を走査して退避されたキャッシュラインのデータとそのアドレスである 0x100 番地を取得する．次に，4 つの int 型データを，ラインアドレスから算出した元のメモリアドレスに対して，それぞれストアする命令を発行するようにプロセッサ・コアに働きかける．このように，アボート操作ではログ領域を走査したり，値の書き戻しに要するオーバヘッドが存在する．しかし，トランザクション実行中に競合が発生しなければトランザクションをアボートする必要がないため，競合の発生が少ないようなプログラムでは，アボートによるオーバヘッドがプログラム全体の実行速度に与える影響は少ない．

3　LogTM の問題点と解決策

本章では，既存研究である LogTM の問題点を指摘し，その問題が性能に与える影響を予備評価する．さらにそれらの問題を解決する方法を探る．

3.1　LogTM の問題点

既存の LogTM では，競合の検査をキャッシュライン単位で行っており，個々の変数に対するアクセスの競合を検出することができない．このため，複数のスレッドが異なるデータにアクセスしたとしても，それらが同一キャッシュライン上に存在していた場合は競合として判定されてしまうという問題がある．

この問題を，**図 5** のように 2 つのスレッドが異なるデータにアクセスする場合を例に説明する．図 5 中の Core と Cache はそれぞれプロセッサ・コアと各コアが持つローカルキャッシュを簡略化して示したものであり，Shared Memory は 2 つのコアに共有されている主記憶を表している．各コア及び主記憶の間を繋ぐ通信網は省略する．また，Core1 及び Core2 にはそれぞれ Thread1 及び Thread2 が割り当てられており，各

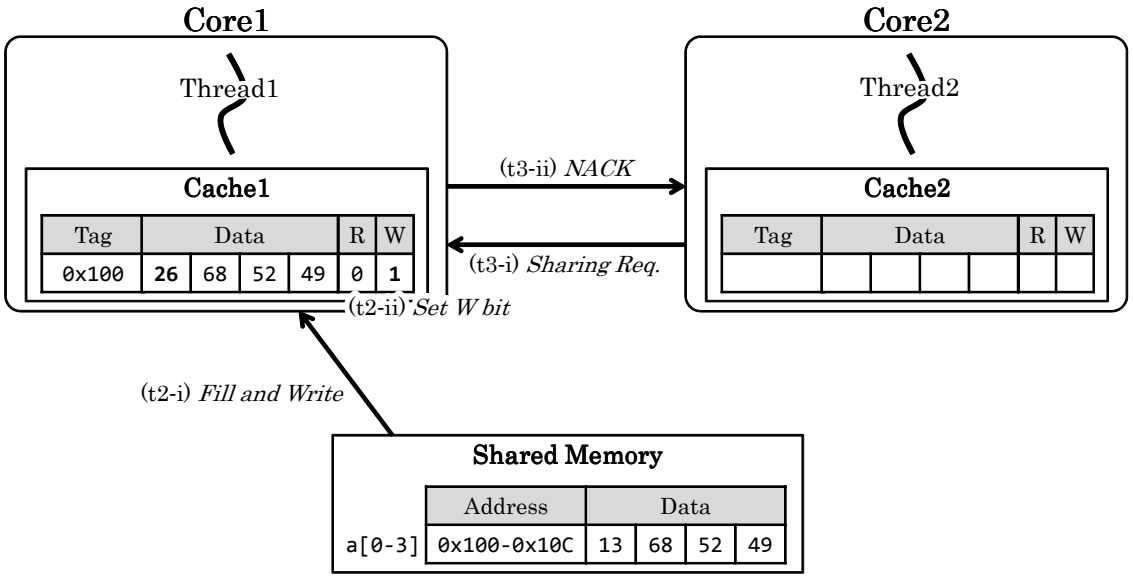

図 5: 異なる変数へのアクセスによる無駄な競合

表 1: プログラム例と実行スケジュール

時刻	Thread1	Thread2
t1	BEGIN_XACT;	BEGIN_XACT;
t2	a[0]=26;	
t3		tmp=a[2];
⋮	⋮	⋮
⋮	COMMIT_XACT;	COMMIT_XACT;

スレッドでは**表 1** に示すようなプログラムが実行されているとする．なお，表 1 中の BEGIN_XACT; 及び COMMIT_XACT; はそれぞれトランザクションの開始及び終了を表しており，トランザクションの範囲を定義している．表 1 中の時刻はプログラム内の各式が実行される時刻 ($t1 < t2 < t3$) を表している．

まず，2 つのスレッドがトランザクションの実行を開始し (時刻 t1)，Thread1 が `a[0]` へ値を代入する (時刻 t2)．このとき，Cache1 には `a[0]` が保持されていないため， 主記憶の 0x100 番地から始まるデータがキャッシュされ，`a[0]` の値が更新される (t2-i)．このとき，アドレス 0x100 及び更新される前のキャッシュラインのデータがログ領域に退避されるが，図中では省略する．また，トランザクション内でライトアクセスが発生したため，0x100 番地に対応するキャッシュラインの write ビットがセットされる

(t2-ii).

次に，Thread2 が a[0] と同一キャッシュライン上に存在する a[2] の読み出しを試みる (時刻 t3)．このとき，Cache2 でも a[2] は保持されていないため，a[2] の値を読み出す必要があり，一貫性プロトコルに従って Cache1 と 0x100 番地のラインを共有するためのリクエストが送信される (t3-i)．リクエストを受け取った Thread1 は 0x100 番地に対応するラインの read 及び write ビットを参照する．このとき，write ビットがセットされているため競合として判定され，Thread2 に対して NACK が返信される (t3-ii).

このように，複数のスレッドがそれぞれ異なるアドレスを持つ変数にアクセスしたとしても，それらの変数が同一キャッシュライン上に存在していた場合，競合はそのラインに割り当てられた read 及び write ビットによって検査されるため，無駄な競合が検出される可能性がある．LogTM では，NACK を受信したすべてのスレッドはストールするため，本来なら競合として判定される必要のないスレッドにも NACK が返信され，ストールしてしまう．本論文では，無駄な競合によって発生したストールを**無駄な競合によるストール**(**false-stall**) と呼ぶ.

この false-stall はトランザクションの範囲内だけでなくトランザクション範囲の外でも観測されることがある．この例を**図 6** を用いて説明する．図中の 2 つのスレッドでは，それぞれ**表 2** に示すようなプログラムが実行される．また，表 2 中の int 型配列 b1 は Thread1 で定義されたスレッドローカルな変数であり，同様に b2 も Thread2 で定義されたスレッドローカルな配列変数である．これらは主記憶上の 0x200 番地以下に配置されており，各要素は同一キャッシュライン上に存在しているとする．

まず，Thread1 が b1[0] に対する代入を行い (時刻 t1)，続いてトランザクションの実行を開始する (時刻 t2)．このとき，Thread2 ではまだトランザクションは実行されていない．次に，Thread1 は時刻 t3 でトランザクション内部で b1[0] を読み出すため，対応する read ビットがセットされ，さらに a[0] に値を代入するため，a[0] に対応するラインの write ビットもセットされる (t3)．その後，Thread2 がトランザクションの範囲外で b2[0] に代入を試みる (時刻 t4)．このとき，Cache2 には b2[0] のデータが保持されていないため， 0x200 番地のラインを無効化するためのリクエストが送信される (t4-i)．このリクエストを受け取った Thread1 では，b1[0] が配置されたラインに対する read after write が発生したことが検知され，競合として判定される．したがって，Thread2 に対して NACK が返信されてしまい (t4-ii)，Thread2 はその実行を停止する． LogTM では，トランザクションを実行するスレッドはすべてのリクエストに

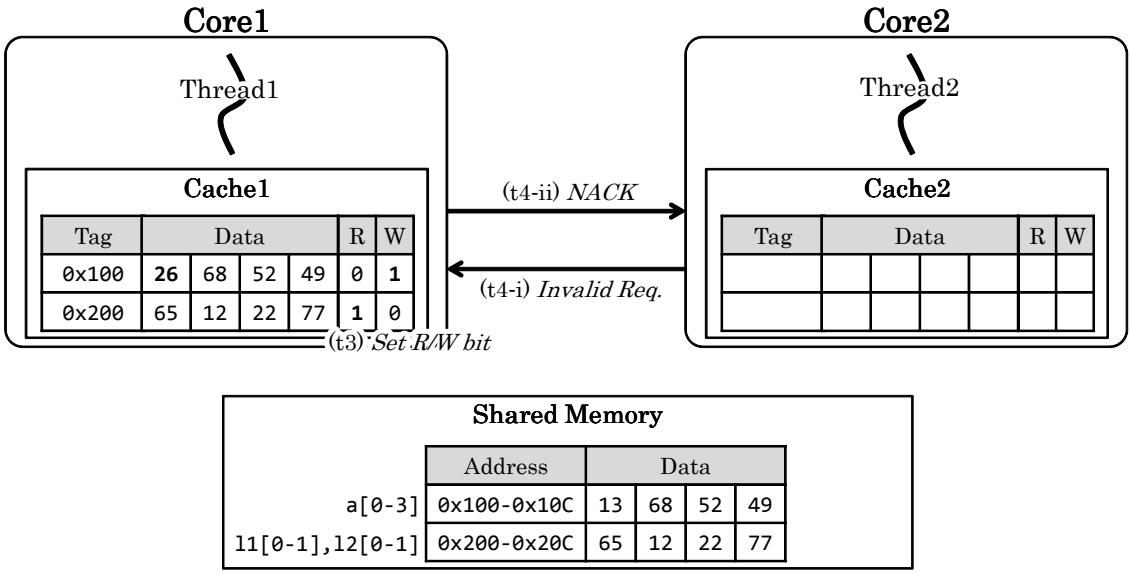

図 6: トランザクション範囲外で発生する無駄な競合

表 2: トランザクション範囲外で false-stall が発生するプログラム例と実行スケジュール

時刻	Thread1	Thread2
t1	b1[0]=31;	
t2	BEGIN_XACT;	
t3	a[0]=b1[0]	
t4		b2[0]=82;

対して競合を検査するため，本来なら競合として検出する必要のないトランザクション外に存在する変数へのアクセスであっても競合と判定され， false-stall が発生する可能性がある．

ここで，同一キャッシュライン上に存在し，なおかつ複数のスレッドによりそれぞれアクセスされた場合に false-stall が引き起こされるようなデータの組み合わせを以下の 5 種類に分類する．

(1) 配列の異なる要素
(2) 構造体の異なるメンバ変数
(3) グローバルに定義された異なる変数
(4) スレッドローカルに定義された異なる変数
(5) グローバル変数とスレッドローカル変数

これまでに説明した 2 つの例のうち，図 5 で説明した例は，同一キャッシュライン上に配置された a[0] と a[2] に 2 つのスレッドがそれぞれアクセスしているため，(1) に該当する．また， 図 6 で説明した例では，配列変数名の異なる 2 つのスレッドローカルな配列 b1 及び b2 が同一ライン上に配置されているため， (4) に該当する．通常，スレッドローカルに宣言された異なる変数は別々のスタック領域上に配置されるため，同一キャッシュライン上に配置されることはない．しかし，それらの変数が malloc() により確保された場合， malloc() は一次元のヒープ領域からメモリ領域を確保するため，スレッドローカルな変数であっても同一キャッシュライン上に存在する可能性がある．したがって，(4) 及び (5) のように，スレッドローカルな変数同士が同一キャッシュライン上に配置されたり，スレッドローカルな変数とグローバルな変数が同一キャッシュライン上に配置されることは十分考えられる．

数名が共通しているため，同一のデータ構造に属している．したがってこのようなデータを本論文では**同一データ構造**と定義し，同一データ構造内の要素の間で発生する false-stall を**同一データ構造による false-stall** と呼ぶ．一方で (3)，(4) 及び (5) のようなデータは，(1) 及び (2) とは異なり，変数名が共通していない．したがってこのようなデータを以降**異なるデータ構造**と呼び，異なるデータ構造の間で発生する false-stall を**異なるデータ構造による false-stall** と呼ぶ．

3.2 予備評価

3.1 節で挙げた 2 つの false-stall が，既存の LogTM 上でどれだけ発生し，それが全体の性能にどの程度影響しているかを，シミュレーションにより予備評価した．ベースとなるプロセッサ構成は 32 個の SPARC V9 プロセッサ・コアを持つ CMP(Chip Multi-Processor) とし， OS は Solaris10 とした．各コアは 2 次元のメッシュ型に接続されており，それぞれに対してプライベートな L1 キャッシュ，すべてのコア間で共有される L2 キャッシュ，及びディレクトリが割り当てられている．このようなシステムを Simics[7](ver 3.0.31) と GEMS[8](ver 2.1.1) を用いてシミュレートした．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．**表 3** に詳細なシミュレーションパラメタを示す．

評価対象のプログラムは STAMP(Stanford Transactional Applications for Multi-Processing)[9] ベンチマークプログラムに含まれる Vacation, Genome に加えて，GEMS 付属の microbench から Prioque，Sortedlist を用い，それぞれのプログラムを 4 及び 8

表 3: シミュレータ諸元

Processor	SPARC V9
#cores	32 cores
clock	4 GHz
issue width	single
issue order	in-order
non-memory IPC	1
L1 I&D cache	32 KBytes
ways	4 ways
latency	1 cycle
line size	64 Bytes
L2 cache	8 MBytes
ways	8 ways
latency	20 cycles
line size	64 Bytes
L2 Directory	Full-bit vector sharers list
latency	6 cycles
Memory	4 GBytes
latency	500 cycles
Interconnect network	2D mesh topology
link latency	3 cycles
link bandwidth	64 Bytes

スレッドで実行した．各プログラムの入力を**表 4** に示す．

予備評価の結果を**図 7** に示す．図中のグラフは，各ベンチマークプログラムと前述のスレッド数との組み合わせによる結果を表している．凡例は得られたサイクル数の内訳を示しており，FALSE-STALL-XACT はトランザクションの範囲内で発生した false-stall， FALSE-STALL-NONXACT はトランザクションの範囲外で発生した false-stall， XACT は FALSE-STALL-XACT 以外のトランザクションの実行，NONXACT は FALSE-STALL-NONXACT 以外のトランザクション範囲外の実行に要したサイクル数をそれぞれ示している．

なお，フルシステムシミュレータ上でマルチスレッドを用いた動作のシミュレーショ

表 4: ベンチマークプログラムとその入力

Vacation	64K entries, 4K tasks, 8 queries, 10 rel, 80 users
Genome	16 length, 256 gene length, 16K seg
Prioque	8192 ops
Sortedlist	500 ops, 64 length

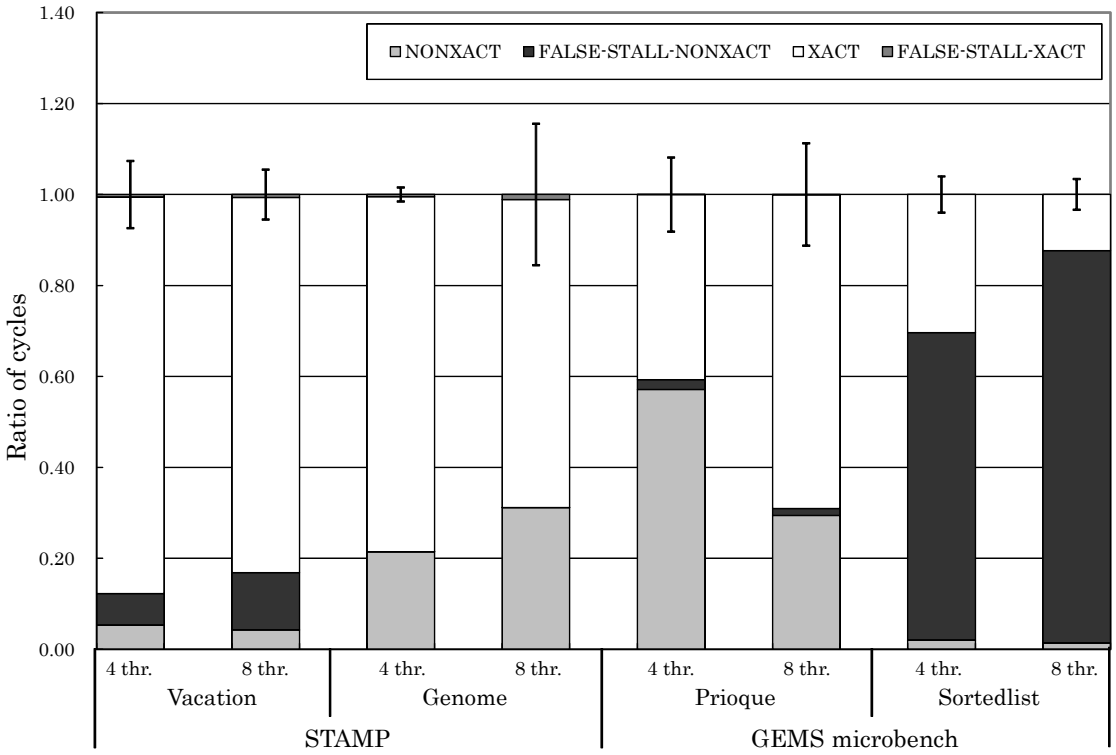

図 7: 予備評価の結果

ンを行う場合は性能のばらつきを考慮しなければならない [10]．したがって，各評価対象につき試行を 10 回繰り返し，得られたサイクル数から平均値と 95%の信頼区間を求めた．信頼区間はグラフ中のエラーバーで表す．

図 7 より，既存の LogTM では全体の実行サイクル数のうち，FALSE-STALL-NONXACT が平均で 22.1%， Sortedlist を 8 スレッド実行した場合に最大で 86.3%発生した．また， FALSE-STALL-XACT が平均で 0.4%， Genome を 8 スレッド実行した場合に最大で 1.1%発生した．このことから，2 種類の false-stall のうち，主に FALSE-STALL-NONXACT が既存の LogTM に悪影響を及ぼしていることがわかる．

ここで，2 つの false-stall について，各ベンチマークプログラムの特徴に基づきそれ

ぞれ見ていく．まず，FALSE-STALL-NONXACT に注目すると，Vacation，Prioque，Sortedlist で有意に発生しており，Genome でもわずかながら発生した．各プログラムで FALSE-STALL-NONXACT を引き起こしたデータの組み合わせは，3.1 節で挙げた false-stall が発生する 5 種類のデータの組み合わせのうちの (3) 及び (4) に該当するパターン，すなわち異なるデータ構造であることがわかった．これは，FALSE-STALL-NONXACT が発生していた変数のメモリ領域がいずれも malloc() によって一次元のヒープ領域から確保されており，グローバルな変数やスレッドローカルな変数でもメモリ領域内の近いアドレスに配置されたためである．ここで，特に他のプログラムに比べて非常に大きな割合を占めている Sortedlist に注目する．Sortedlist ではソートされた単一のリスト構造がランダムに検索されるが，グローバルに定義されたリスト構造の一部とライブラリ関数 srandom() 内で確保された共有構造体が同一キャッシュライン上に配置されており，トランザクション範囲外で random() によりその構造体がアクセスされたときに FALSE-STALL-NONXACT が観測されている．これは false-stall 発生パターン (3) に該当する．FALSE-STALL-NONXACT が発生したスレッドは，相手のトランザクションが終了するまで停止するが，Sortedlist で実行されるトランザクションは実行時間が長いため，FALSE-STALL-NONXACT の占める割合が大きくなったと考えられる．また，Prioque では，ひとつの一次元配列と構造体が各スレッドで共有されており，それらはメモリ空間上の近い位置に配置されているため，同一キャッシュライン上にそれらのデータの一部が存在する可能性がある．したがって，Sortedlist と同様に false-stall 発生パターン (3) に該当する．このように，Sortedlist と Prioque は特定のデータ構造が同一ライン上に存在することで false-stall が発生する．一方で，Vacation では false-stall を引き起こすデータ構造が複数存在する．それらは各スレッドでローカルに確保された複数の配列であり，メモリ空間上の近い位置に混在している．その結果，異なるスレッドにより確保された異なる配列が同一ライン上に存在していたため，パターン (4) に該当し，FALSE-STALL-NONXACT が発生している．

次に，FALSE-STALL-XACT に注目すると，Vacation，Genome，Prioque で発生している．しかし，FALSE-STALL-NONXACT に比べて総実行サイクル数に占める割合は小さい．これは，FALSE-STALL-NONXACT が発生したことでスレッドの実行が停止し，同時に実行されるトランザクション数が制限されたことが原因であると考えられる．また， これら 3 つのプログラムで FALSE-STALL-XACT が発生したデータの組み合わせはパターン (1) 及び (2) に該当した．例えば，Prioque では，共有された一次元配列の要素のうち同一キャッシュライン上に存在する異なる要素に対して複数の

スレッドがアクセスしていたため，パターン (1) に該当し false-stall が発生している．また，Vacation では，各スレッドが木構造を共有しており，それぞれのノードを構成する構造体のメンバ変数が同一キャッシュライン上に存在していた．その構造体の異なるメンバ変数が複数のスレッドによりアクセスされていたため，パターン (2) に該当する．

また，今回評価対象としたプログラムでは false-stall 発生パターン (5) が確認されなかった．しかし，一次元のヒープ領域からメモリ領域を切り出すような malloc() を利用した場合，パターン (5) に該当するデータ対が同一ライン上に存在する可能性は十分ある．

これまでに挙げた 2 つの false-stall の発生を防止するためには，競合の検出をキャッシュライン単位ではなく，ライン上に存在する個別のデータ単位で行うように細粒度化する必要がある．それを実現するための方法として，以下の 2 つが考えられる．

方針 (a) 異なるデータを別々のキャッシュラインへ分散させる

false-stall 発生の原因となる可能性のあるデータをそれぞれ異なるキャッシュラインに分散して配置させる．これによりキャッシュライン単位で競合を検査した場合でも，アドレスの異なるデータが同一ライン上に存在することがなくなる．

方針 (b) メモリ上に配置されたデータ単位で競合を検出する

キャッシュライン単位ではなく，メモリ上に配置されたデータ単位で競合を検査する．これにより異なる変数及び要素に対するメモリアクセスを判別することができるため，個別のデータに対する競合を正しく判定できる．

以下， 4 章及び 5 章でこれら 2 つの実現方法に基づいた手法をそれぞれ提案する．

4 異なるキャッシュラインへの分散配置

本章では，方針 (a) に基づき，変数を異なるキャッシュラインへ分散して配置する手法を提案し，その実装方法について説明する．以降，本章で提案する手法を**提案手法 1** と呼ぶ．

4.1 変数の分散配置モデル

提案する変数の分散配置モデルにおいて，分散の対象とする変数と，その動作モデルについて説明する．

4.1.1 分散の対象とする変数

通常，プログラマは共有メモリ空間上の同一変数に関しては，複数スレッドからのアクセスの並行性を意識的に制御するが，異なる変数に関しては並行性制御の対象から除外してプログラミングする．しかし，複数の変数が同一キャッシュライン上に存在し，それらの変数に複数のスレッドがアクセスした場合，3.1 節で説明したように false-stall が発生する可能性がある．このような false-stall の発生を防ぐためには，false-stall を発生させうるすべての変数を，異なるキャッシュラインへ配置するプログラムの記述が必要となる．しかし，そのようなプログラムを設計するためには，プログラマは利用するアーキテクチャの知識を持つ必要がある．また，同一キャッシュライン上に存在すると false-stall が発生するような変数を判別する必要があるため，プログラマの負担となりうる．

そこで，本研究では，複数のスレッドが並列にアクセスするキャッシュライン上に配置される複数の変数を，別々のキャッシュラインへ自動的に分散配置する手法を提案する．これにより，プログラマに異なる変数間で競合が発生する可能性を意識させることなく false-stall の発生を防止することができる．

この方法ついて，**図 8** に示すように，2 つの構造体 `itemA` 及び `itemB` が malloc() により確保される場合を例に説明する．これら 2 つの構造体は構造体変数名が異なるため，異なるデータ構造である．既存の LogTM では，これらの構造体は**図 9**(a) のように同一キャッシュライン上に配置される可能性がある．すると，異なるデータ構造による false-stall が発生する．そこで，提案手法ではこれらの構造体を図 9(b) に示すように異なるラインへ分散して配置させることで，false-stall の発生を防止することができる．

しかし，このような配置を行った場合，2 つの構造体が割り当てられたメモリ領域の間に無駄な空間が存在するため，既存手法に比べてデータの局所性が低下し，同時にフラグメンテーションも発生する可能性が高まる．さらに，空間的に連続する配列の要素にアクセスするようなプログラムに対して適用した場合，データの局所性が大きく低減することでキャッシュミスも増大する危険性がある．

これについて，**図 10** に示す配列 `array` の要素を分散させる場合を例に説明する．この配列は，既存手法では**図 11**(a) のように配置される．この配列に複数のスレッドがアクセスした場合，同一データ構造による false-stall が発生する可能性がある．ここで，この配列が持つ 4 つの要素を図 11(b) のように異なるラインへ分散して配置することが考えられる．これにより，同一データ構造による false-stall の発生は防止できる

```
1  typedef struct item{
2    int code;
3    int price;
4  }item_t
5  item_t* item_A = (item_t*)malloc(sizeof(item_t));
6  item_t* item_B = (item_t*)malloc(sizeof(item_t));
7
8  void thread1(void){
9    int local=0;
10   BEGIN_XACT;
11         local = item_A.code + item_A.price;
12   COMMIT_XACT;
13 }
14 void thread2(void){
15   int local=0;
16   BEGIN_XACT;
17         local = item_B.code + item_B.price;
18   COMMIT_XACT;
19 }
```

図 8: 2 つのスレッドが構造体にアクセス

Address	Data			
0x100-0x10C	item_A .code	item_A .price	item_B .code	Item_B .price

(a) 既存手法

Address	Data			
0x100-0x10C	item_A .code	item_A .price	-	-
0x110-0x11C	item_B .code	Item_B .price	-	-

(b) 分散配置した場合

図 9: 2 つの構造体のメモリ配置

```
1  #define NUM_ELEMS 6
2
3  int* array;
4  array = (int*)malloc(NUM_ELEMS * sizeof(int));
5
6  void thread1(void){
7    int local=0;
8    BEGIN_XACT;
9        local = array[0] * 2;
10   COMMIT_XACT;
11 }
12 void thread2(void){
13   int local=0;
14   BEGIN_XACT;
15       local = array[2] + 2;
16   COMMIT_XACT;
17 }
```

図 10: 2 つのスレッドが配列にアクセス

Address	Data			
0x200-0x20C	array [0]	array [1]	array [2]	array [3]

(a) 既存手法

Address	Data			
0x200-0x20C	array [0]	-	-	-
0x210-0x21C	array [1]	-	-	-
0x220-0x22C	array [2]	-	-	-
0x230-0x23C	array [3]	-	-	-

(b) 分散配置した場合

図 11: 配列のメモリ配置

が，データの局所性が大きく低減する．

このように，データを分散配置する手法には，false-stall の発生を防止できるが，メモリの利用効率を悪化させてしまうという欠点がある．特に，同一データ構造の各要素

に対して適用した場合に顕著である．したがって，false-stall を防止できる割合とデータ局所性はトレードオフな関係にある．しかし，3.2 節の予備評価の結果より，異なるデータ構造による false-stall に比べて同一データ構造による false-stall は小さい．したがって，本提案手法の分散対象となる変数を異なるデータ構造のみに限定する．これにより，メモリ利用効率の低下とフラグメンテーションの発生をある程度抑制しつつ false-stall を削減できると考えられる．

さらに，3.1 節でも説明したように，グローバルな変数やスレッドローカルな変数は，malloc() によって確保された場合にメモリ空間上の近い位置に配置され，同一キャッシュライン上に存在する可能性がある．そこで，本提案手法では malloc() により確保される変数を分散の対象とする．

4.1.2 メモリ領域の効率的利用

配列の各要素や構造体の各メンバ変数を異なるキャッシュラインへ分散させることに比べ，異なるデータ構造を別々のラインへ分散させる場合は，データの局所性の低減率が小さいが，依然としてそれらの変数の間に無駄な空間が存在している．そこで，このような無駄なメモリ領域に対して，同一ライン上に存在したとしても false-stall が発生しないような変数を配置することで，メモリ利用効率及びデータ局所性が低減することの防止を目指す．

これについて，**図 12** に示すようなサンプルプログラムをマルチスレッド実行する場合を例に説明する．このプログラムが実行されると，グローバルに定義された配列 `array`，Thread1 でローカルに定義された構造体 `item_A` 及び `item_B`，そして Thread2 上で定義されたローカルな構造体 `item_C` 及び `item_D` がメモリに配置される．これらの変数が，既存手法では**図 13**(a) のように配置されたとする．この場合，Thread2 が図 12 の 30 行目を実行することで更新したキャッシュライン 0x310 は，Thread1 により 15 行目で更新される可能性がある．このため，異なるデータ構造による false-stall が発生してしまう可能性がある．また，Thread2 が 31 行目でライン 0x320 上の `item_C.price` を読み出した後に，Thread1 が 16 行目において `item_B.code` を更新することでも false-stall が発生する可能性がある．

前者のような false-stall は，配列 `array` と構造体 `item_A` という異なるデータ構造が同一ライン上に存在したことで発生する．一方で後者のような false-stall は，異なるスレッドで定義された構造体同士が同一ライン上に存在したことで発生する．したがって，これら false-stall 発生の原因となった変数をそれぞれ異なるラインへ配置させることで false-stall の発生を防止できる．

```
1  #define NUM_ELEMS 6
2
3  typedef struct item{
4    int code;
5    int price;
6  }item_t
7
8  int* array;
9  array = (int*)malloc(NUM_ELEMS * sizeof(int));
10
11 void Thread1(void){
12   item_t* item_A = (item_t*)malloc(sizeof(item_t));
13   item_t* item_B = (item_t*)malloc(sizeof(item_t));
14
15   item_A.code = 11; item_A.price = 35;
16   item_B.code = 22; item_B.price = 46;
17   BEGIN_XACT;
18         array[0] = item_A.code + item_B.code;
19         array[1] = item_A.price + item_B.price;
20   COMMIT_XACT;
21 }
22
23 void Thread2(void){
24   item_t* item_C = (item_t*)malloc(sizeof(item_t));
25   item_t* item_D = (item_t*)malloc(sizeof(item_t));
26
27   item_C.code = 94; item_C.price = 65;
28   item_D.code = 16; item_D.price = 32;
29   BEGIN_XACT;
30         array[4] = item_C.code + item_D.code;
31         array[5] = item_C.price + item_D.price;
32   COMMIT_XACT;
33 }
```

図 12: マルチスレッド実行されるサンプルプログラム

Address	Data			
0x300-0x30C	array [0]	array [1]	array [2]	array [3]
0x310-0x31C	array [4]	array [5]	item_A .code	item_A .price
0x320-0x32C	Item_B .code	item_B .price	item_C .code	item_C .price
0x330-0x33C	item_D .code	item_D .price	-	-

(a) 既存手法

Address	Data			
0x300-0x30C	array [0]	array [1]	array [2]	array [3]
0x310-0x31C	array [4]	array [5]	-	-
0x320-0x32C	item_A .code	item_A .price	Item_B .code	item_B .price
0x330-0x33C	item_C .code	item_C .price	item_D .code	item_D .price

(b) 最適な分散

図 13: メモリ配置例

ここで，これらの構造体を分散配置させたときに発生する無駄な空間を有効的に利用するために，false-stallが発生しないような変数をその空間に挿入する．例えば，item_A と item_B が同一ライン上に配置されたとしても，そのラインが複数のスレッドからアクセスされることはないため，item_B と同じラインに item_A を挿入する．同様に，item_C と item_D も同一ラインに配置することができる．結果として，図 13(b) のように変数を配置させることで，false-stall の発生を防止しつつ，フラグメンテーションの

発生及びメモリの利用効率の低下をある程度抑えることができる．

以上より，提案する分散配置方法は以下の手順に従うものとする．

1. **グローバルな変数同士を分散**
   3.1 節で述べた false-stall 発生パターンのうち，パターン (3) に該当することで発生する false-stall を削減する．
2. **異なるスレッドにより定義されたローカル変数を分散**
   false-stall 発生パターン (4) に該当する false-stall を削減する．
3. **グローバルな変数とスレッドローカルな変数を分散**
   false-stall 発生パターン (5) に該当する false-stall を削減する．
4. **同一スレッドにより定義されたローカル変数を同一キャッシュライン上に挿入**
   変数の分散により発生した空きメモリ領域を利用することで，フラグメンテーションの発生を抑え，メモリの利用効率を向上させる．

4.2 malloc のラップによるキャッシュライン・パディング

4.1 節で提案した分散配置モデルを簡略化したものとして， 4.1.1 項で述べた，異なるデータ構造を別々のキャッシュラインに分散して配置するモデルの実装について説明する．

4.2.1 簡易モデルの概要

簡易モデルでは，malloc() により確保される変数を分散の対象とし，その変数が必要とするメモリ領域のサイズがキャッシュラインサイズの倍数となるように無意味なデータを挿入する．これにより，同一ライン上に異なるデータ構造が存在することを防ぐ．

ここで，図 12 で示したプログラムに対して本手法を適用した場合のメモリ状態を**図 14** に示す．配列 `array` は int 型の要素を 6 つ持っており，$4 \times 6 = 24$ バイトのメモリ領域を確保する必要がある．キャッシュラインサイズが 16 バイトである場合，簡易モデルでは配列 `array` がキャッシュラインサイズの倍数である 32 バイトとなるようにパディングされる．同様に，構造体 `item_A`，`item_B`，`item_C` 及び `item_D` が格納されるメモリ領域も，それぞれの合計サイズがキャッシュラインサイズの倍数になるようにパディングされる．このように，異なるデータ構造が同一キャッシュライン上に配置されないため，異なるデータ構造による false-stall の発生を防止できる．

4.2.2 ラッパー関数によるパディング

4.2.1 項で説明した簡易モデルを実現する場合，確保するメモリ領域サイズを計算し，

Address	Data			
0x300-0x30C	array [0]	array [1]	array [2]	array [3]
0x310-0x31C	array [4]	array [5]	PAD	PAD
0x320-0x32C	item_A .code	item_A .price	PAD	PAD
0x330-0x33C	Item_B .code	item_B .price	PAD	PAD
0x340-0x34C	item_C .code	item_C .price	PAD	PAD
0x350-0x35C	item_D .code	item_D .price	PAD	PAD

図 14: パディング適用後のメモリ配置例

```
1  #define LINESIZE 16
2
3  size_t padded_size = ((size+LINESIZE-1) & ~(LINESIZE-1));
```

図 15: メモリ領域サイズの算出

それにより算出されたサイズを malloc() に渡す操作が必要となる．そこで，malloc() の操作に対して前処理を挿入するラッパー関数を定義する．このラッパー関数のオブジェクトを malloc() 呼出し時にプリロードすることで，簡易モデルを実現する．

まず，ラッパー関数内部では，確保したいメモリ領域サイズを算出する．**図 15** は実装したラッパー関数の一部を表す．図 15 中の変数 `size` は既存の malloc() によって動的に確保されるメモリ領域のサイズを表しており，`padded_size` はそれをキャッシュラインサイズの倍数となるようにパディングしたサイズを表す．また，キャッシュラインサイズは `LINESIZE` として関数内部で定義する．このような計算式により，パディング後のメモリ領域サイズを取得できる．

次に，取得したメモリ領域のサイズを既存の malloc() に渡す操作を行う．この操作を，ラッパー関数が定義された共有ライブラリを作成し，それをダイナミックリンクされた既存の malloc() と置き換えることで実現する．共有ライブラリ内部では，malloc とは異なる名前の関数を新たに作成し，その関数のポインタに既存の malloc() の関数ポインタを格納することで，既存の malloc() を退避する．新たに作成した関数に対して，図 15 で取得したメモリ領域のサイズを引数として渡すことで，目的のサイズのメモリ領域を確保することができる．

表 5: プログラム例と実行スケジュール

時刻	Thread1	Thread2
t1	BEGIN_XACT;	BEGIN_XACT;
t2	a[0]=26;	
t3		a[2]=70;
t4	COMMIT_XACT;	

5 変数単位による競合検出

本章では方針 (b) に基づいた手法である変数単位による競合検出手法を提案し、その実装方法について述べる。

5.1 提案する競合検出

本節では，提案する変数単位による競合検出を実現するための方法を，競合の検査及びアボートという 2 つの観点から考える．以降，本節で説明する手法を**提案手法 2** と呼ぶ.

5.1.1 競合の検査

既存の LogTM はリクエストを受け取った時にキャッシュライン単位で競合の検査を行っている．そこで，提案手法ではリクエストの受信時に，メモリ操作によりアクセスされる個々のデータ単位で競合を検査する.

提案する競合の検査手法について，**表 5** に示すトランザクションを含むプログラムを， **図 16**(a) に示す 2 つのスレッドがそれぞれ実行する例を用いて説明する．このとき，配列 a は 0x100 番地に配置され，その要素が同一キャッシュライン上に存在しているとする．このプログラムが実行されると，まず 2 つのスレッドはトランザクションを開始し (時刻 t1)，続いて Thread1 が a[0] へ値を代入する (時刻 t2)．すると，0x100 番地のデータがキャッシュされ，a[0] の値が更新される (図 16(a)(t2))．ここで，提案手法ではアクセスした a[0] へのアクセス情報のみが保持されるとする.

次に，Thread2 が a[2] へ値の代入を試みる (時刻 t3)．すると，Thread1 が持つ 0x100 番地のキャッシュラインを無効化するリクエストが送信される (t3-i)．このとき，リクエストには無効化するキャッシュラインのラインアドレス 0x100 が保持されているが，アクセスする a[2] のアドレスは保持されていない．そのため，Thread1 は a[2] に対するアクセスの競合を検査することができない．そこで，提案手法ではリクエストを

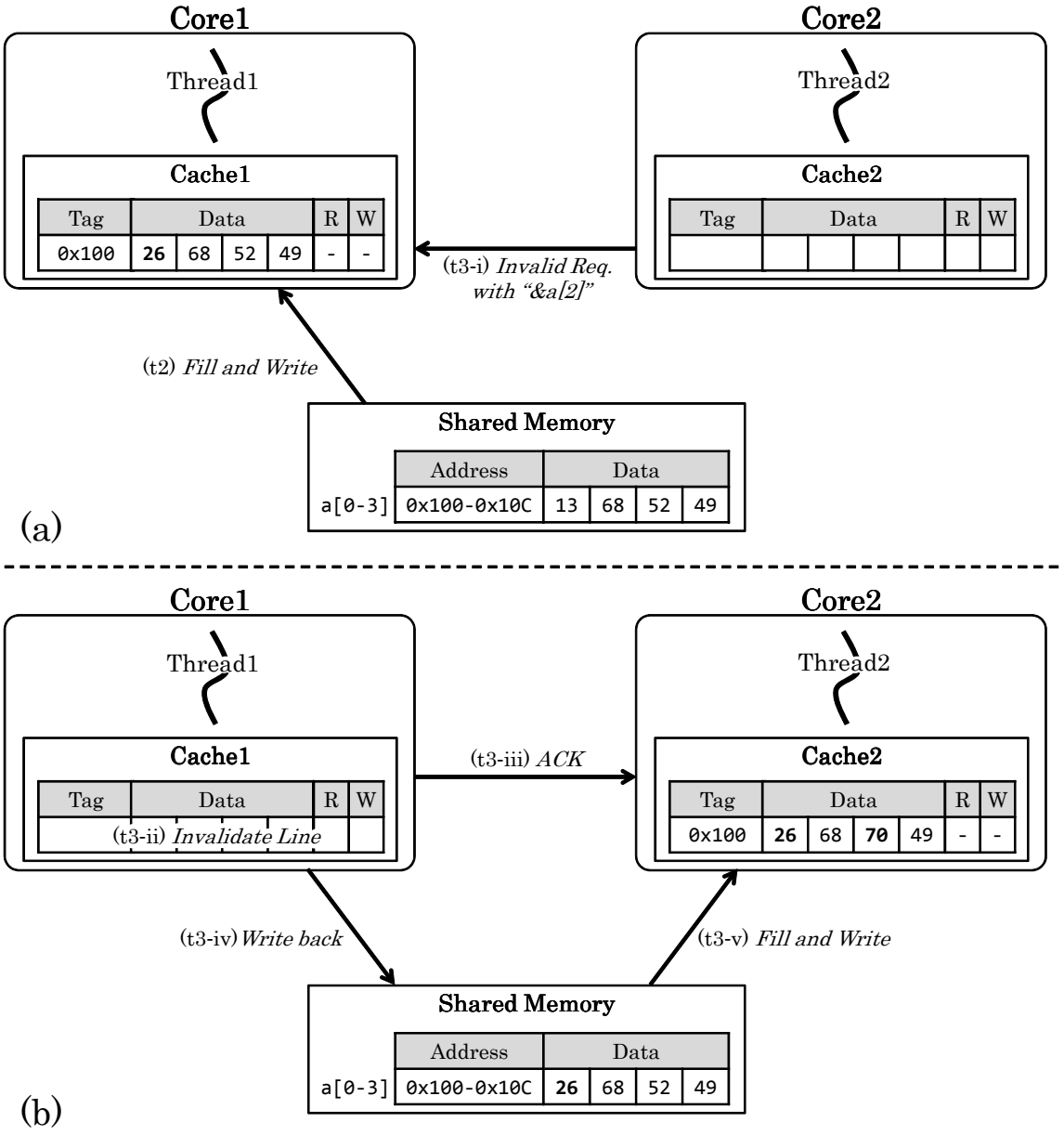

図 16: 同一キャッシュライン上の変数アクセスを許可

送信すると同時に，a[2] のアドレスも Thread1 へ送信する．これにより，リクエストと a[2] のアドレスを受け取った Thread1 は保持していた変数のアクセス情報を参照し，a[2] との競合を検査することができる．このとき，Thread1 は a[2] へリード及びライトアクセスしていないため，Thread2 が行った a[2] へのアクセスとは競合し

ないと判定される．したがって，Thread1 は既存の一貫性プロトコルに従って，0x100 番地のキャッシュラインを無効化し (図 16(b)(t3-ii))，Thread2 に対して ACK を送信し (t3-iii)，キャッシュラインをライトバックする (t3-iv)．その後，Thread2 は 0x100 番地のラインをキャッシュし，a[2] の値を更新する (t3-v)．

5.1.2 アボート時の操作

トランザクションをアボートさせる場合，ログに退避された値を使用することで，トランザクション内で更新した値を更新前の状態に復元する必要がある．このとき，ログ領域にはキャッシュラインのデータが退避されており，既存の LogTM では退避されたラインのデータが全て復元される．このため，他のスレッドで更新された同一ライン上の変数の値も，ログに退避された値で上書きされてしまう．これは，既存の LogTM ではキャッシュライン単位で他のスレッドからのアクセスを制御しており，トランザクション中でライトアクセスされたラインが他のスレッドにアクセスされることが考慮されていないためである．そこで，提案手法では自身のトランザクションで更新した値のみを書き戻すことで，キャッシュの一貫性を保持する．

これについて，**図 17** を例に説明する．図 17 は，図 16 の例における表 5 時刻 t3 で発生した a[2] への代入が完了した直後のプロセッサ及びメモリ状態を表しており，Thread1 のログ領域には表 5 の時刻 t2 の直前の時点におけるキャッシュラインのデータが保持されている．ここで，Thread1 がトランザクションをアボートしたとする．このとき，ログ領域に退避されたラインのデータを全て書き戻してしまうと，Thread2 で更新された a[2] の値が上書きされてしまい，キャッシュの一貫性が保持できない．このため，Thread1 は自身のトランザクション内で更新した a[0] の値のみを元のアドレスに書き戻すことで，Thread2 で更新された a[2] の値を維持する．さて， 2.2.2 項で述べたように，トランザクションがアボートされると，アボートハンドラによりログ領域に退避された各値を元のメモリアドレスにそれぞれストアする命令が実行される．そこで，a[0] に対するストア命令のみを実行させる (i)．このとき，Cache1 には 0x100 番地のキャッシュラインが保持されていないため，0x100 番地のラインを無効化するリクエストが Thread1 から送信される (ii)．リクエストを受け取った Thread2 は 0x100 番地のキャッシュラインを無効化し (iii)，Thread1 へ ACK を返信し (iv)，キャッシュラインをライトバックする (v)．ACK を受け取った Thread1 は，0x100 番地のラインをキャッシュし，ログ領域に退避されていた値を用いて a[0] を復元する (vi)．このとき，主記憶には復元後の a[0] の値が反映されていないが，今後 0x100 番地のキャッシュラインが共有または無効化されるときに主記憶に書き戻されるため，値の一貫性は保証され

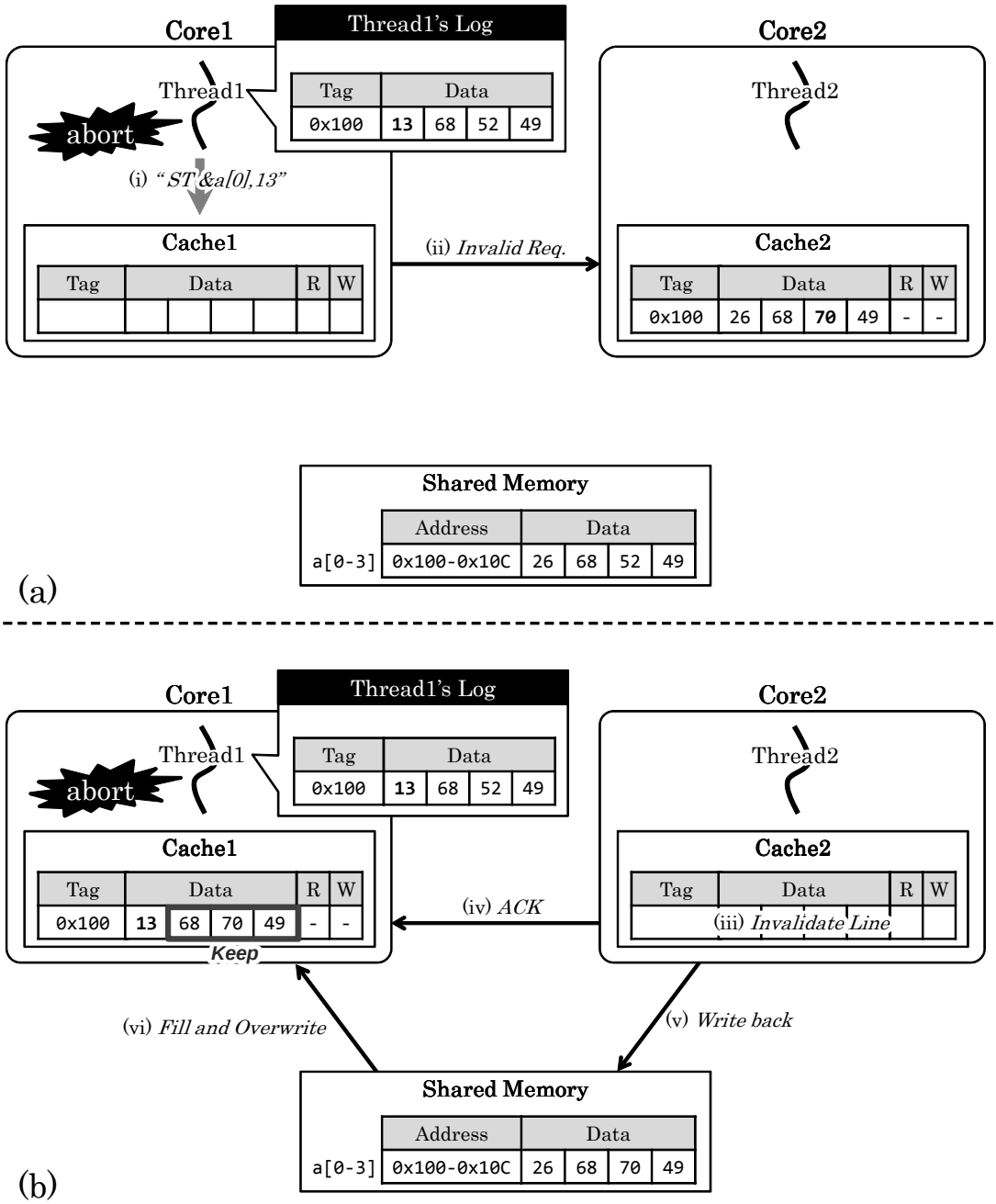

図 17: 更新した値のみ書き戻し

る．結果として，a[0] の値のみを復元することができる．

5.2 実装

本節では 5.1 で提案した競合検出方法を実装するにあたって必要となるハードウェア及びその動作について説明する．

5.2.1 ハードウェア拡張

変数単位によって競合を検査する機構を実装する最も簡単な方法は，各キャッシュラインに対する read 及び write ビットの数を，ライン上に存在し得る変数の数だけ用意することである．しかし，競合が発生しない変数のアクセス情報は記憶する必要がないため，拡張したハードウェアの全てが使用されるとは限らない．したがって，この方法ではハードウェアの無駄が大きい．

そこで，追加するハードウェア量を削減するために，変数単位で競合を検査する必要があるキャッシュラインに限定して，リード・ライトアクセス情報を記憶するための少容量から成るハードウェアを追加する．これを**R/W テーブル**と呼ぶ．また，キャッシュラインから R/W テーブルへ一意にアクセスできるように，R/W テーブルのインデクスを保持する **Ptr** フィールドをキャッシュの各ラインに追加する．

拡張するハードウェア構成を**図 18** に示す．R/W テーブルでは，既存手法で競合検査の対象であったキャッシュラインというメモリブロックを，提案手法では N 個に分割し，分割後の各要素を競合検査の最小単位として扱う．これにより，ひとつのキャッシュライン上に存在する N 個のデータに対する競合を検査することができる．この R/W テーブルは，プロセッサ・コアごとに 1 つ保持しており，キャッシュタグを記憶する Tag，リードアクセス情報を記憶する R，自コアに割り当てられたスレッドが実行するトランザクション内のライトアクセス情報を記憶する W^{my}，他スレッドで実行されるトランザクション内で発生したライトアクセス情報を記憶する $\mathrm{W}^{\mathrm{other}}$ フィールドを持つ．R，W^{my} 及び $\mathrm{W}^{\mathrm{other}}$ は，それぞれのアクセス情報を保持するための N ビットのレジスタにより構成される．また，R/W テーブルで扱う単位当たりのメモリサイズを縮小する，すなわち N を増大させることで，それだけ多くのデータに対するアクセス情報を保持することができるが，必要なハードウェア量が増大する．そのため，キャッシュラインサイズと検査対象とするデータサイズを考慮してアーキテクチャを設計する必要がある．例えば，16 バイトのキャッシュライン上に int 型のデータが 4 つ保持される場合を考える．ここで，$N = 4$ としたとき，提案手法では 4 つのデータのアクセス情報を全て記憶できる．しかし，$N = 2$ とした場合，$N = 4$ の場合と比べて，ハードウェア量は削減されるが，4 つのデータに対する全てのアクセス情報を保持できないため，false-stall が生じる可能性が高まる．また，$N = 8$ とした場合は，ハード

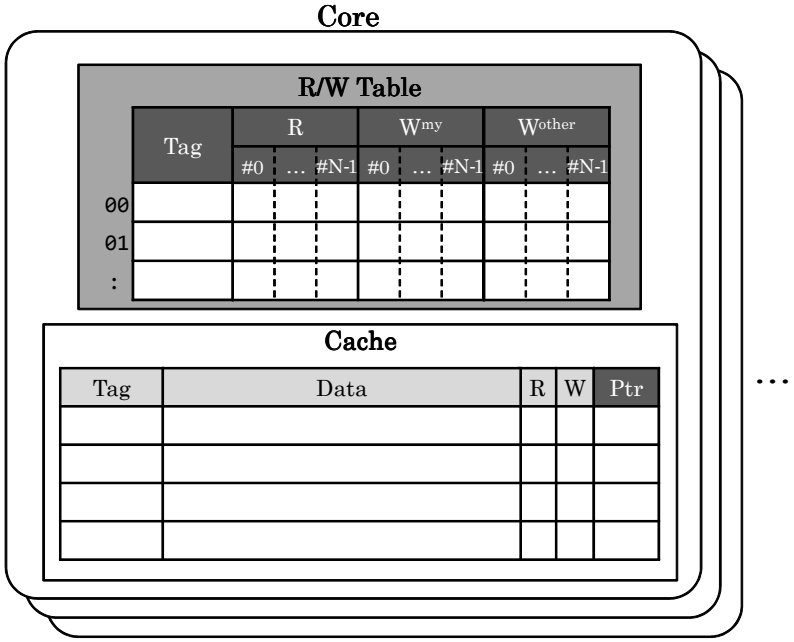

図 18: 提案するアーキテクチャ構成

ウェア量が増大するうえ，保持する必要のあるアクセス情報は 4 つで十分であるため，R/W テーブルのうち 50%しか利用されない．したがって，この例の場合は $N = 4$ とした場合が最も適切であると考えられる．

ここで，キャッシュラインサイズが L バイトのとき，ある変数のアドレス $addr$ に対してトランザクション内でリードアクセスが発生する場合，セットされる R のエントリ R[#n] のインデクス n を

$$n = \left\lfloor \frac{addr \bmod L}{L/N} \right\rfloor \tag{1}$$

として計算する．同様に，セットされる W^{my} 及び $\mathrm{W}^{\mathrm{other}}$ の要素も式 (1) により算出される．

5.2.2 R/W テーブルを利用した競合の検出

提案手法では変数単位で競合を検査したいキャッシュラインを R/W テーブルによって管理する．目的のキャッシュラインを変数単位で競合検査する動作を R/W テーブルに対する 5 つの操作に基づき説明する．

R/W テーブルへのポインタ追加

既存手法では，トランザクション内で発生したすべてのキャッシュラインに対するメモリアクセス情報を read 及び write ビットにより保持していたが，提案手法では競

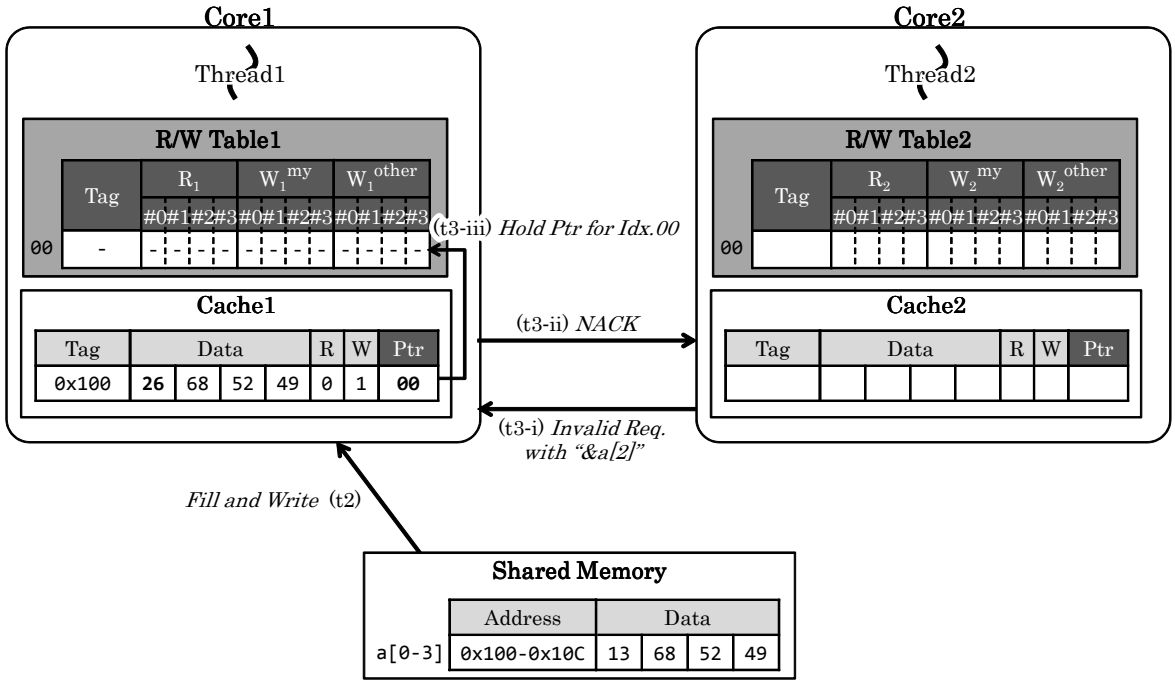

図 19: 競合発生時に R/W テーブルへのポインタを保持

合が発生したラインのみを R/W テーブルで管理する．しかし，事前に競合が発生するキャッシュラインを判定することは，プログラムを解析し，競合する可能性のある変数を検出する必要があるため困難である．そこで，提案手法では過去に一度でも競合が発生したキャッシュラインを R/W テーブルで管理する対象とすることで，静的解析などに必要なコストを削減する．ただし，この方法では，最初に発生した競合に限り，そのアクセスを既存手法と同様にライン単位で管理するため，false-stall が発生する可能性がある．しかし，2 回目以降も同じキャッシュラインにおいて競合が発生すれば，そのラインは変数単位で競合を判定できるため，性能に大きな影響を与えないと考えられる．

この提案アーキテクチャ上で，表 5 で示されるような，配列の異なる要素にそれぞれアクセスするトランザクションを実行する例を**図 19** に示す．なお，キャッシュラインサイズ $L = 16$，$N = 4$ であるとする．まず，2 つのスレッドでトランザクションの実行が開始される (時刻 t1)．その後に Thread1 上で `a[0]` への代入が実行され (時刻 t2)，`a[0]` の値が更新される (t2)．このとき，0x100 番地のラインに対する競合が過去に発生していないとすると，既存手法と同様にライン上の write ビットがセットされる．

次に，Thread2 が `a[2]` への代入を試みる (時刻 t3) が，0x100 番地からのキャッシュ

ラインは既に Thread1 のキャッシュに保持されているため，ラインの無効化リクエストが送信される．このとき，送信されるリクエストに対して a[2] のアドレスが付加される (t3-i)．しかし，この時点では，0x100 番地に対する競合はキャッシュライン単位で検査されているため，write ビットにより競合と判定され，Thread2 へ NACK が返信される (t3-ii)．ここで，0x100 番地に対する競合が検出されたため，これ以降 0x100 番地のラインを R/W テーブルを用いて管理する必要がある．そのため，0x100 番地のラインの Ptr に R/W テーブルのインデクスがセットされる (t3-iii)．この場合，時刻 t3 の時点では，Thread1 は過去にアクセスした変数のアドレスを保持していないため，キャッシュライン上のどの変数に対応する R_0，W_0^{my} 及び W_0^{other} の要素をセットする必要があるのかを判断できない．したがって，この時点では R/W テーブルのエントリに対して値はセットされない．

R/W テーブルのエントリ登録

前述したように，R/W テーブルで競合を管理すると判定された瞬間には，過去にそのライン上のどの変数がアクセスされたかを判別することができない．そこで，判定時に実行されているトランザクションでは，既存手法と同様にライン単位で競合を検査し，それ以降に実行されるトランザクション中で発生した変数へのアクセス情報を，R/W テーブルで記憶する．

R/W テーブルへのポインタをセットしたトランザクションが終了した場合の動作を，先ほどの図 19 の例を用いて説明する．この例では，時刻 t3 以降も，Thread1 は 0x100 番地から成るキャッシュラインに対してライン単位で競合を検査する．ここで，Thread1 で実行されるトランザクションがコミットされたとすると (時刻 t4)，既存手法と同様に read 及び write ビットがクリアされる．しかし，コミット以降も 0x100 番地のキャッシュラインに対して変数単位で競合を検査するためには，このラインで過去に競合が発生したという情報を保持する必要がある．したがって，Thread1 がコミットした場合，0x100 番地の Ptr の値 00 は維持される．

続いて，コミットした Thread1 が別のトランザクションを実行する例を**図 20** に示す．図 20 中の Thread1 及び Thread2 はそれぞれ**表 6** のトランザクションを実行するとする．説明の都合上，Thread3 及びそれが割り当てられているコアの図を省略する．まず，3 つのスレッドでトランザクションの実行が開始される (時刻 t5)．その後に Thread1 上で a[0] への代入が実行され (時刻 t6)，a[0] の値が更新される (t6-i)．このとき，0x100 番地のキャッシュラインの Ptr がセットされているため，このラインに対する競合が過去に発生しており，これ以降 R/W テーブルにより競合を検査する必要があることがわ

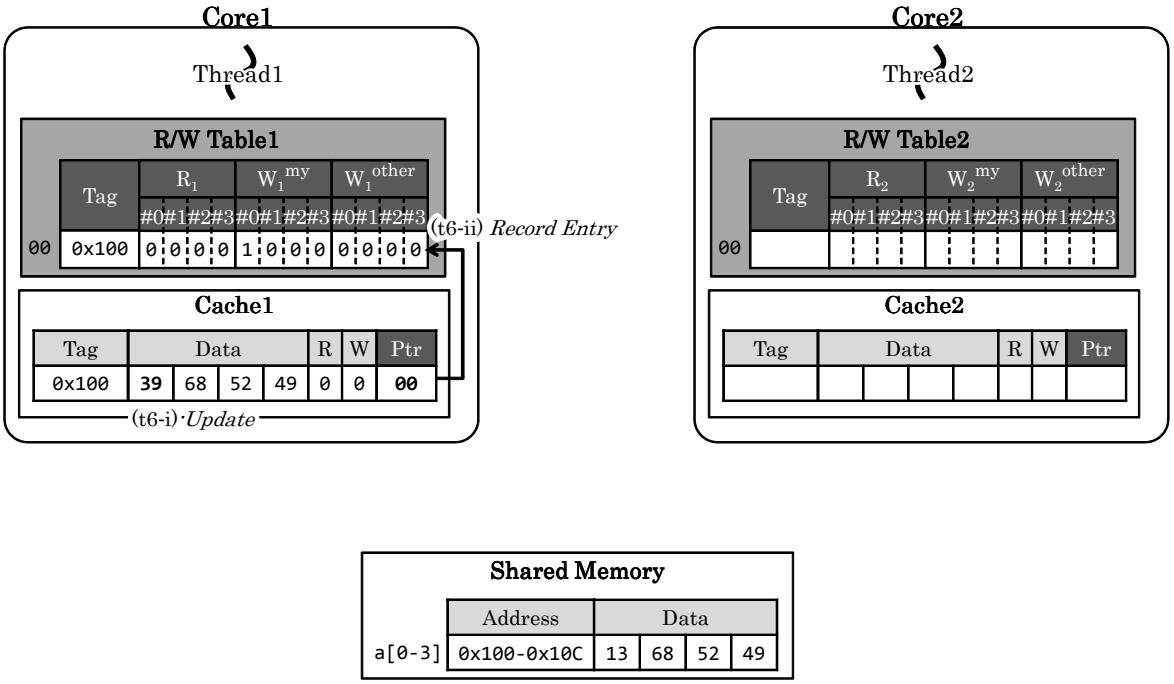

図 20: R/W テーブルのエントリを挿入

表 6: プログラム例と実行スケジュール 2

時刻	Thread1	Thread2	Thread3
t5	BEGIN_XACT;	BEGIN_XACT;	BEGIN_XACT;
t6	a[0]=39;		
t7		a[2]=70;	
t8			tmp=a[0]
t9		COMMIT_XACT;	

かる．そこで，R/W テーブルのインデクス 00 に対してエントリを挿入する (t6-ii)．挿入するエントリにはラインの持つ Tag がセットされる．そして，アクセス情報を記憶させるために，変数に対応した R_0，W_0^{my} フィールドをセットする．この例では，Thread1 は a[0] にライトアクセスするため，W_1^{my}[#0] にのみ 1 がセットされ，残りは 0 がセットされる．また，この時点ではまだ 0x100 番地のキャッシュラインは他のトランザクションにアクセスされていないため，W_1^{other} フィールドには 0 がセットされる．

R/W テーブルのエントリ参照による競合の検査

リクエストを受け取ったときに，そのラインに対応する R/W テーブルエントリが

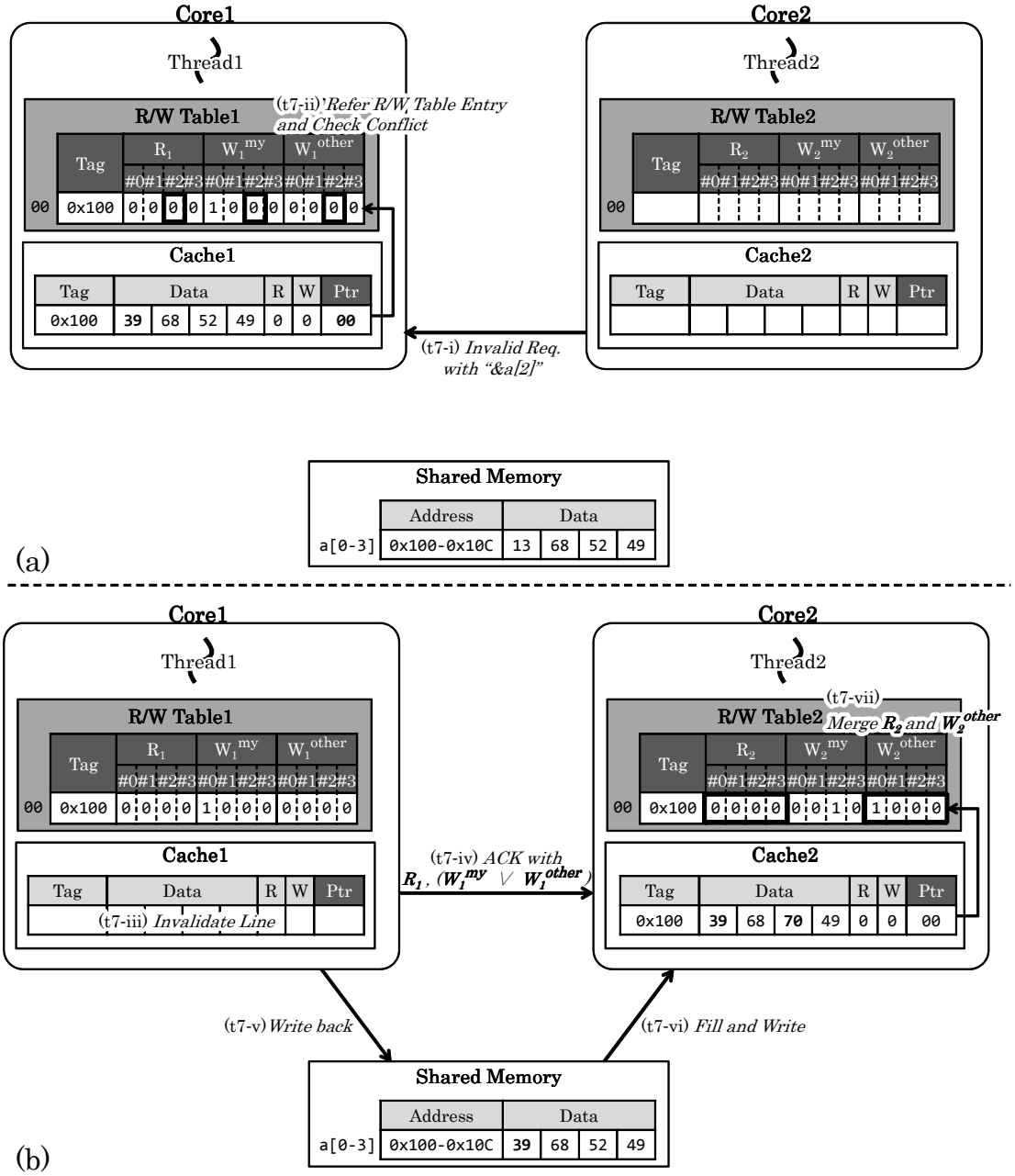

図 21: R/W テーブルを利用した競合の検査

存在していたならば，変数単位で競合を検査する．前述した表 6 のトランザクションを実行する例を再び考える．現在，時刻 t6 まで実行が完了しているとする．続いて，Thread2 が a[2] に対して値の代入を試みる (時刻 t7)．このとき，**図 21**(a) で示されるように，a[2] のアドレスを付加した無効化リクエストが Thread2 から送信される

(t7-i)．ここで，リクエストを受け取ったThread1のR/Wテーブルには，0x100番地のラインに対応するエントリが存在しているため，そのエントリを参照することで競合を検査する(t7-ii)．リクエストには`a[2]`のアドレスが付加されているため，式(1)により`a[2]`に対応する位置を知ることができる．このとき，`a[2]`に対応している$R_1[\#2]$，$W_1^{my}[\#2]$及び$W_1^{other}[\#2]$には0がセットされているため，競合は検出されない．したがって，Thread1は0x100番地を無効化し(図21(b)(t7-iii))，同時に0x100番地のPtrもクリアする．しかし，R/Wテーブルのエントリはクリアされずに維持される．これにより，もし今後Cache1が再び0x100番地のキャッシュラインを保持したならば，R/Wテーブルは0x100番地のTagを記憶しているため，該当するR/Wテーブルのインデクスを Ptrにセットできる．

さて，競合の検査はリクエストを受信したスレッドが行うが，0x100番地のキャッシュラインが無効化されたため，今後0x100番地に対するリクエストがThread1に送信されることがなくなり，Thread1は0x100番地上に存在する変数に対する競合を検査することができなくなる．そこで，Thread2はThread1のトランザクション内におけるリード・ライトアクセス情報を保持することで，Thread1における競合を検査する責任を代わりに請け負う．そのため，Thread1はThread2に返信するACKに対して，0x100番地のラインに対応するR/WテーブルのR_1フィールドの値，及びW_1^{my}とW_1^{other}フィールドの論理和を求めた値を付加する(t7-iv)．以降，W_1^{my}とW_1^{other}フィールドの論理和を求めた値をW_1と呼ぶ．このACKが送信された後，Thread1は0x100番地のラインを書き戻す(t7-v)．

この後，Thread2はACKによりそのラインにアクセスできるようになったことを知るため，そのラインをキャッシュする(t7-vi)．ここで，Thread2はACKに付加されたR_1及びW_1により，0x100番地が過去に競合が発生したラインであることを知るため，0x100番地に対するR/Wテーブルエントリが挿入され，そのインデクスをPtrに保持し，`a[2]`に対するライトアクセス情報を$W_2^{my}[\#3]$に記憶する．ここで，Thread1のトランザクションで過去に行われたリード・ライトアクセス情報を保持するために，R_2と受信したR_1の論理和及びW_2^{other}とW_1の論理和を求め，それぞれをR_2とW_2^{other}に上書きする(t7-vii)．これにより，以降他のトランザクション内で発生した`a[0]`へのアクセスに対する競合をThread2が検査することができる．

ここで，Thread3が`a[0]`を読み出す操作を試みるとする(表6時刻t8)．Thread3は**図22**に示されるようにCore3に割り当てられている．なお，Core3内部のキャッシュ及びR/Wテーブルは図では省略している．まず，`a[0]`のデータをキャッシュ上に保

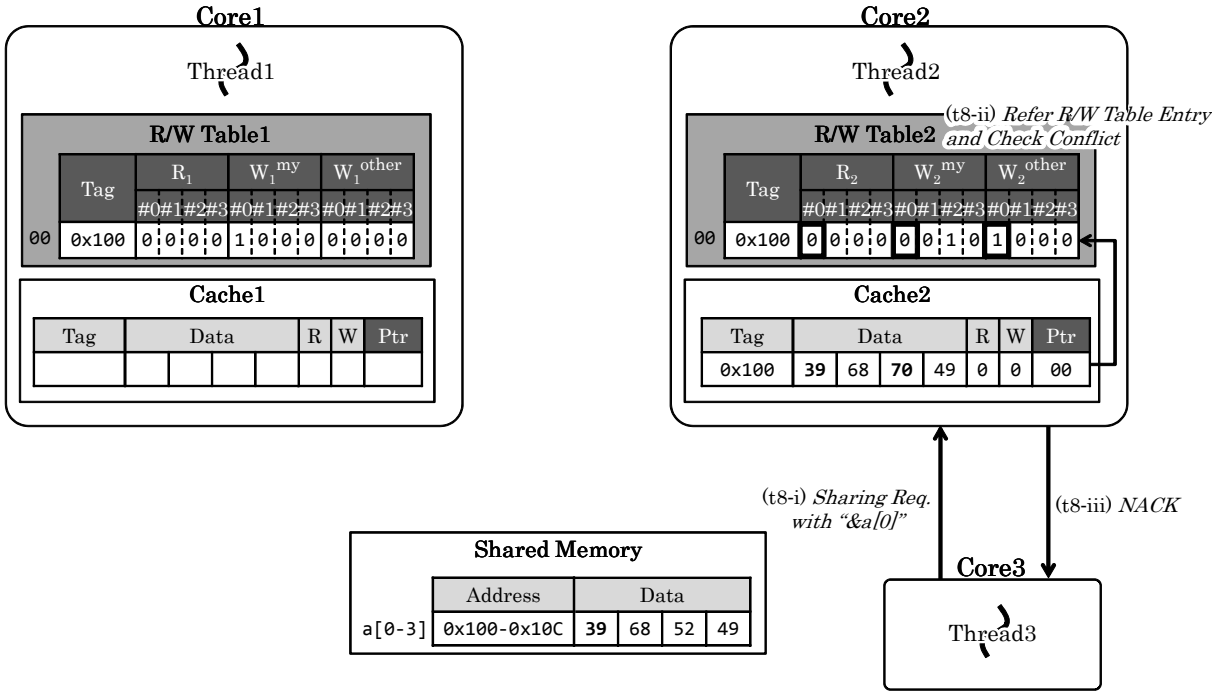

図 22: Thread2 による競合の検査

持しているのは Thread2 のみであるため，Thread3 は Thread2 に対して 0x100 番地のキャッシュラインを共有するためのリクエストを，`a[0]` のアドレスが付加された状態で送信する (t8-i)．リクエストを受信した Thread2 は，`a[0]` のアドレスから式 (1) により算出したインデクスを元にして R/W テーブル上の $R_2[\#0]$，$W_2^{my}[\#0]$ 及び $W_2^{other}[\#0]$ を参照することで競合を検査する (t8-ii)．このとき，$W_2^{other}[\#0]$ に 1 がセットされているため，Thread2 以外のスレッドが過去に `a[0]` へライトアクセスしたことがわかる．したがって，このアクセスは競合として判定され，Thread3 に対して NACK が返信される (t8-iii)．

R/W テーブルの破棄

R/W テーブルに記憶されるリード・ライトアクセス情報は，実行中のトランザクション内で発生した固有の情報であるため，そのトランザクションが終了した場合は，R/W テーブルに記憶されたアクセス情報を全て破棄する必要がある．そのため，トランザクションがコミット及びアボートされた場合は R/W テーブル のエントリを全て削除する．

これについて，表 6 の時刻 t9 において，Thread2 で実行されているトランザクションがコミットされた場合を例に説明する．このとき，R/W Table2 が持つエントリは，

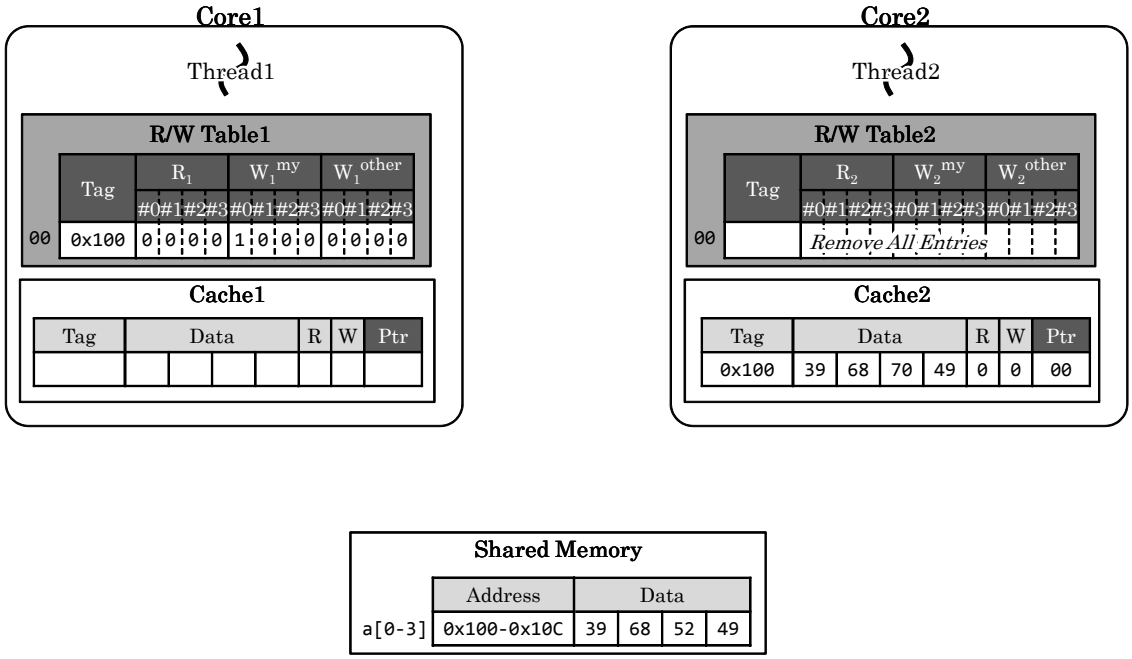

図 23: コミット後

図 23 のように全て削除される．一方で，Ptr フィールドはクリアされずに維持される．この操作は，トランザクションがアボートされたときも同様に行われる．

R/W テーブルのオーバーフロー

提案手法を適用した場合，一度でも競合が検出されたキャッシュラインは，それ以降 R/W テーブルによって管理される．そのようなラインはキャッシュ上から追い出されたとしても，その情報が他スレッドに対して伝搬されるため，永続的に R/W テーブルへエントリが登録される．このため，プログラムの実行が完了するまでに一度しか競合と判定されないキャッシュラインでも R/W テーブルで管理され続ける．しかしながら，R/W テーブルの容量は有限であるため，そのような不要なエントリを保持し続けることによってオーバーフローが起こりやすくなってしまう．

そこで，不要なエントリを R/W テーブル上から削除するため，R/W テーブルがオーバーフローした場合に，そのトランザクションをアボートさせる．同時に，R/W テーブルのエントリに対応するキャッシュラインの Ptr を全てクリアする．これらの操作により，不要なエントリを全て削除することができる．一方で，今後も競合する可能性のあるキャッシュラインに対応するエントリもすべて破棄されてしまうが，そのような有用なエントリは再び競合が発生した場合に R/W テーブルで管理される．結果

として，今後競合する可能性の無いキャッシュラインのみを R/W テーブル の管理下から除外し，不要なエントリにより R/W テーブルが圧迫されることを防ぐことができる．

また，R/W テーブルのオーバーフローによりそのスレッドで管理されている R/W テーブルはクリアされるが，他スレッドの R/W テーブルには今後競合が発生しないキャッシュラインに対応するエントリが残っている可能性がある．しかし，そのようなキャッシュラインは他スレッドからはアクセスされないため，そのラインを R/W テーブルで管理しているかどうかの情報が他スレッドに伝搬されることはない．したがって，あるスレッドの R/W テーブルにそのようなラインのエントリが存在していたとしても，そのキャッシュラインが R/W テーブルによって管理されていたという情報は R/W テーブルが破棄された時点で削除されるため，特別な操作は必要ない．

5.2.3 R/W テーブルを利用した書き戻し対象データの限定

5.1.2 項で述べたように，アボート時における書き戻し対象はアボートされるトランザクション内で更新した変数のみとする必要がある．これを実現するために，R/W テーブルの W^{my} フィールドを利用する．W^{my} には，実行されているトランザクション内で，どの変数に対してライトアクセスが発生したかが記憶されている．そこで，ログに退避された更新前のキャッシュライン状態に対して，そのラインに対応する R/W テーブルエントリの W^{my} をマスキングすることで，書き戻す必要のある値を判別する．

例えば， 5.2.2 項で説明した図 21 の例における Thread1 が，表 5 の時刻 t9 以降にアボートされたとする．Thread1 のログには，**図 24** で示されるような 0x100 番地のキャッシュライン状態が保持されている．このとき，Thread1 の実行するトランザクション内部で発生したライトアクセス情報は W_1^{my} に記憶されている．そこで，アボートにより書き戻すラインのデータに対して，W_1^{my} フィールドの値により図 24 のようにマスクすることで，Thread1 のトランザクション内で更新された `a[0]` の値のみを復元することができる．

このような方法を既存のアボートハンドラを拡張することで実装する．既存のアボートハンドラでは，ログから更新前のデータ及びそのメモリアドレスを読み出し，読み出した値を元のメモリアドレスへストアしている．拡張したアボートハンドラでは，このログからアドレスを読み出す操作を行う際に，そのアドレスに対応する W^{my} の値を R/W テーブルから取得する．取得した W^{my} の値が 1 であるならばログに退避された値を書き戻し，そうでないならば書き戻しを実行しない．このようにして，提案するアボート操作を実現する．

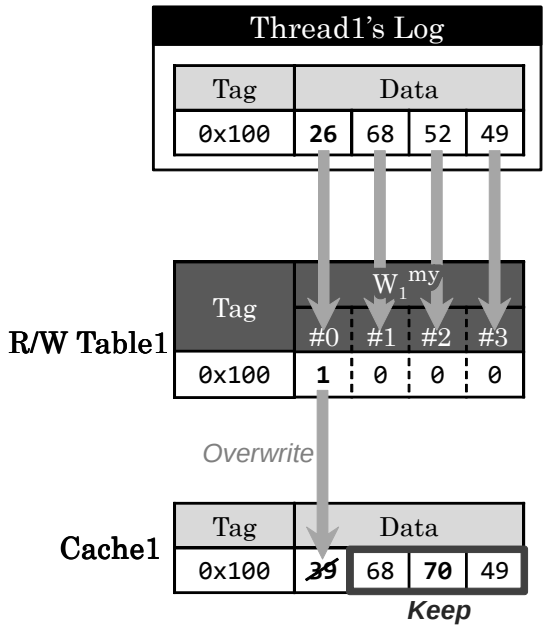

図 24: W^{my} による書き戻し対象の限定

5.3 変数分散手法との融合

5.1 節で説明した本提案手法を適用した場合，全てのメモリ操作でアクセスされるデータに対するリード・ライトアクセス情報を記憶するため，全ての false-stall の発生を防ぐことができる．しかし， 5.2 節で説明したように，既存手法において競合と判定された全てのラインを R/W テーブルを用いて管理するため，競合と判定されるラインが多いほど R/W テーブルのサイズが肥大化する．そこで，本提案手法を 4 章で提案した変数の分散配置手法と融合した手法も考えられる．

提案手法 1 では，異なるデータ構造を別々のキャッシュラインへ分散させることで，異なるデータ構造が同一ライン上に存在した場合に発生する無駄な競合を防止していた．このとき，本手法に対して提案手法 1 を融合させると，異なるデータ構造により発生する無駄な競合は提案手法 1 により防止することができるため，提案手法 2 のみを適用した場合と比べて R/W テーブルに登録されるエントリの数を削減することができる．したがって，提案手法 2 で得られる性能を維持しつつ，その実現に必要なハードウェアコストを削減することができると考えられる．

6 評価

これまでに述べた 2 つの提案手法及びそれらを融合させた手法の速度向上とハードウェアコストを見積もるため，シミュレーションにより評価した．

6.1 評価環境

想定するシステムモデル及び使用ベンチマークプログラムは 3.2 節で述べたものに準ずる．また，各ベンチマークプログラムおよび 4.2.2 項で説明したラッパー関数は，Solaris10 に付属する，単一ヒープ領域からメモリ領域を確保する標準 malloc() 関数を用いて実装し，GNU C コンパイラ (ver 3.4) によりコンパイルした．

なお，提案手法 2 における Prioque 及び Sortedlist に関しては，ソースコードを手動で変換することで，これらのプログラム上の変数が配置されるメモリアドレスが既存手法を適用した場合と同一となるように設定している．提案手法 2 では，プログラムをコンパイルする際に動的にリンクされる，既存手法が提供するソフトウェア・ライブラリに対して，提案手法の実現に必要な一部の操作を追加実装する必要がある．そのため，実行バイナリのコード領域が広がり，データ領域及びヒープ領域に利用されるアドレス空間に既存モデルとのずれが生じた．さて，これら 2 つのプログラムでは， 3.2 節で説明したように，ある特定のグローバルな異なるデータ構造が同一キャッシュライン上に存在し，それらに対して複数のスレッドからアクセスされたことで false-stall が発生する．そのため，それらのデータ構造が配置されるメモリアドレスに少しのずれが生じただけで，同一キャッシュライン上に配置されず，false-stall の発生の仕方に変化が生じる．つまり，これらのプログラムは利用するコンパイラやリンクするライブラリ等によってメモリ配置の状況及び false-stall の発生確率が著しく変動しやすいものであると言える．そこで，既存手法と正確に比較するために，そのずれを修正し，既存手法と同一のメモリ配置を再現した．具体的には，2 つのプログラム内で最初に確保される共有構造体において，その先頭に挿入されているパディング領域を，生じたずれの分だけ削除することで実現した．

6.1.1 評価結果

図 25 に各ベンチマークプログラムを実行した時の評価結果を示す．結果は各ベンチマークプログラムを 4 スレッド及び 8 スレッドで実行したものであり，各ベンチマークプログラムとスレッド数との組み合わせによる結果がそれぞれ 5 本のグラフで表されている．5 本のグラフはそれぞれ左から順に

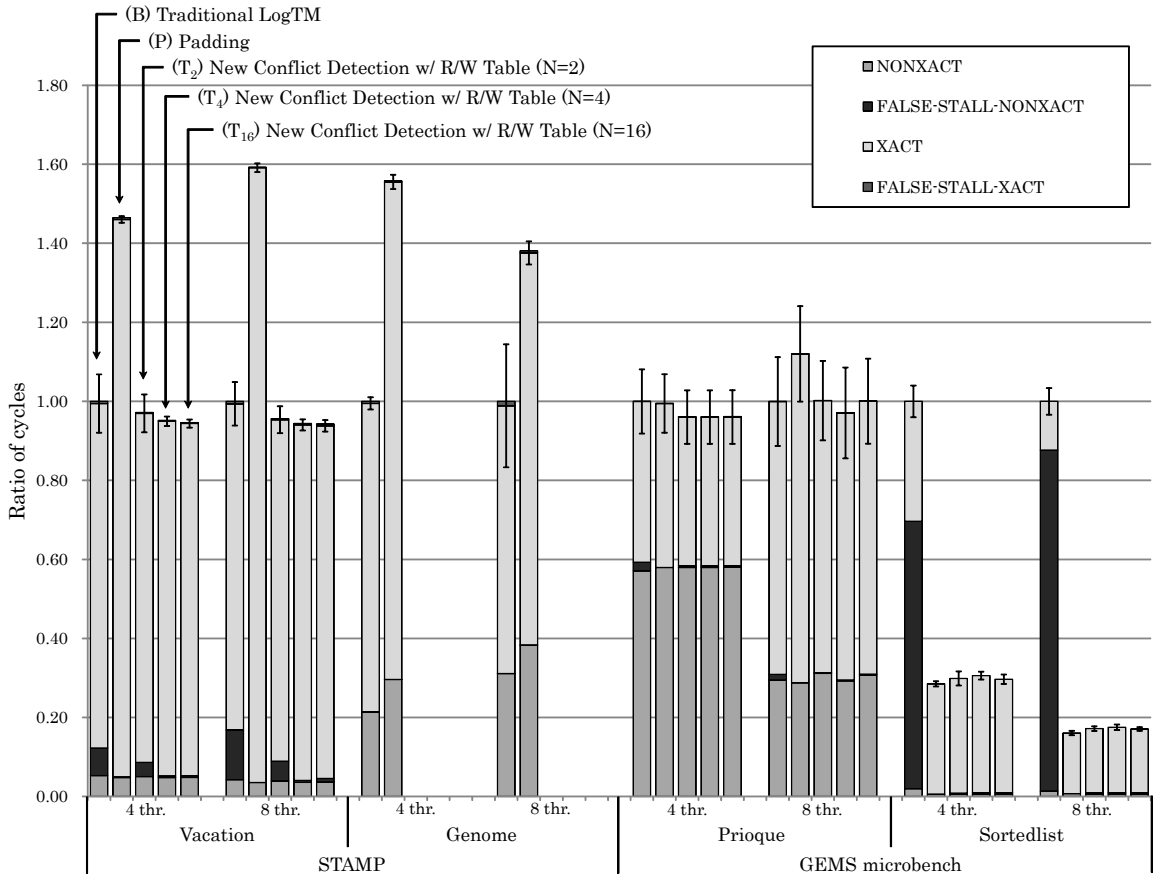

図 25: 評価結果

(B) 既存モデル (ベースライン)

(P) パディングを行う提案手法 1 の簡易モデル

($\mathbf{T}_2$) R/W テーブルを用いて競合を検出する提案手法 2 のモデル ($N = 2$)

($\mathbf{T}_4$) R/W テーブルを用いて競合を検出する提案手法 2 のモデル ($N = 4$)

($\mathbf{T}_{16}$) R/W テーブルを用いて競合を検出する提案手法 2 のモデル ($N = 16$)

の実行に要した総サイクル数を表し，各サイクル数は (B) の実行サイクル数を 1 として正規化しており，凡例は 3.2 節と同様に実行サイクル数の内訳を示している．提案手法 2 に関して，既存手法で競合検査の対象であったキャッシュラインというメモリブロックを，提案手法 2 では N 個に分割し，分割後の各要素を競合検査の最小単位として扱う．また，R/W テーブルのサイズは無制限とし，R/W テーブルのオーバーフローを考慮せずに評価した．

ここで，各提案モデルの結果を見ていく．まず，提案モデル (P) では，全てのベンチ

マークプログラムにおける FALSE-STALL-NONXACT が削減されていることがわかる．また，Sortedlist では大幅な実行サイクル数の削減が認められ，8 スレッドで実行した場合に最大で 83.9%削減された．さらに，Prioque を 4 スレッド実行した場合も，わずかであるがサイクル数が削減された．しかし，残りのベンチマークプログラムでは，FALSE-STALL-NONXACT が削減されたにも拘わらず，総実行サイクル数は削減されなかった．結果，提案モデル (P) における総実行サイクル数は，平均で 7.0%増大し，Vacation を 8 スレッド実行した場合に最悪 59.3%増大となった．

次に，R/W テーブルを用いたモデルのうち，提案モデル (T_{16}) では，Vacation，Prioque 及び Sortedlist でサイクル数の削減が認められる．また，これらの 3 つのベンチマークにおいて FALSE-STALL-NONXACT が削減されていることがわかる．一方で，Genome では，プログラムの動作中に OS からの割り込みが発生し，TLB ミスハンドリング等の処理を長時間実行したため，正常な評価結果を得ることができなかった．そのため，グラフには表記していない．結果，提案モデル (T_{16}) において，Genome を除き，平均で 28.0%，Sortedlist を 8 スレッド実行した場合に最大で 82.9%の実行サイクル数が削減された．

また，R/W テーブルを用いたモデルの中でも提案モデル (T_4)，すなわちキャッシュラインを 4 等分した領域を競合検査の単位とした場合，提案モデル (T_{16}) と同様に，3 つのベンチマークにおいて FALSE-STALL-NONXACT が削減されていることが確認でき，提案モデル (T_{16}) とほぼ同じ総実行サイクル数となった．結果として，提案モデル (T_4) では既存モデル (B) と比べて平均 28.2%，Sortedlist/8 スレッドの場合に最大 82.5%の実行サイクル数が削減された．

一方で，キャッシュラインを 2 等分したブロックを競合検査の単位とした提案モデル (T_2) では，Vacation において提案モデル (T_{16}) 及び提案モデル (T_4) ほどの FALSE-STALL-NONXACT の削減は見られない．しかし，Prioque 及び Sortedlist に関しては，提案モデル (T_{16}) と同等の FALSE-STALL-NONXACT の削減が確認でき，結果として既存モデル (B) と比べて平均 27.3%，Sortedlist/8 スレッドの場合に最大 82.8%の実行サイクル数が削減された．

6.2 考察

各提案モデルについて，その結果及び傾向を詳細に見ていく．

表 7: 提案モデル (P) におけるキャッシュミスペナルティサイクル数の増大率

bench	/thrs	L1 penalty cycles	L2 penalty cycles
Vacation	/4	5.7%	55.7%
	/8	0.7%	52.9%
Genome	/4	71.0%	53.4%
	/8	37.1%	41.4%
Prioque	/4	-3.2%	1.4%
	/8	11.2%	-3.7%

提案モデル (P)

まず，提案モデル (P) では，全てのベンチマークプログラムにおいて，期待通り FALSE-STALL-NONXACT の削減が達成できた．これは，各ベンチマークで FALSE-STALL-NONXACT の発生が見られ，提案手法によりその false-stall の発生を防止することができたためである．特に，Sortedlist では，FALSE-STALL-NONXACT が既存モデル (B) の性能に多大な影響を及ぼしていたため，提案手法による FALSE-STALL-NONXACT の削減に伴って総実行サイクル数も大幅に削減された．

しかし，残りのベンチマークプログラムでは，FALSE-STALL-NONXACT が削減されたにも拘わらず性能の向上が見られなかった．特に，Vacation では，既存モデル (B) において最大で 12.6%も存在していた FALSE-STALL-NONXACT が削減されたが，大幅に性能が低下した．サイクル数の内訳を見ると，XACT 及び NONXACT が増大する傾向が見られる．ここで，これら 3 つのプログラムにおけるキャッシュミスペナルティサイクル数の平均を，既存モデル (B) と比較した場合の増大率を**表 7** に示す．表より，Vacation 及び Genome では，既存モデル (B) に比べてミスペナルティサイクル数が大きく増大する傾向にあった．これは，本提案手法を適用したことでメモリ領域上のデータがパディングされ，空間的局所性が低下したためであると考えられる．特に，Vacation では複数の木構造を各スレッドで共有しており，空間的に連続した木のノードがアクセスされるため，局所性が低下した状況下では木構造の初期参照ミス等が増大する．その結果，XACT 及び NONXACT が増大したと考えられる．したがって，このようなキャッシュミスの増大を抑制するために，4.1.2 項で説明したメモリ領域を効率的に利用するための措置が必要であると言える．

一方で，Prioque では，XACT が増大する傾向があるが，Vacation と Genome ほどのキャッシュミス回数及びミスペナルティサイクル数の増大は見られない．したがっ

表 8: 提案モデル (P) におけるストールサイクル数の増大率

bench	/thrs	Stalled cycles
Vacation	/4	131.5%
	/8	116.1%
Genome	/4	121.2%
	/8	56.6%
Prioque	/4	35.4%
	/8	41.6%

て，キャッシュミス以外に XACT が増大した原因があると推測できる．ここで，3 つのプログラム内で発生したストールサイクル数を既存モデル (B) と比較した場合の増大率を**表 8** に示す．表より，既存モデル (B) に比べてストールサイクル数が増大していることがわかるため，提案モデル (P) では競合の発生する割合が増大していると言える．これは，本提案手法により FALSE-STALL-NONXACT の発生を防止したことで，トランザクション外でスレッドが停止することがなくなり，同時に実行するトランザクション数が増加したためであると考えられる．

提案モデル ($\mathbf{T_{16}}$)

まず，FALSE-STALL-NONXACT について注目すると，Genome を除くほぼ全てのベンチマークにおいて，期待通り大幅に削減されていることがわかる．提案モデル (T_{16}) における FALSE-STALL-NONXACT は，最初に発生した競合が既存モデル (B) と同様にキャッシュライン単位で検査されたために発生するが，そのサイクル数は既存モデル (B) に比べて平均 7.9%まで抑えることができた．したがって，初回に発生する無駄な競合を解消せず，次回以降に防止する方法は十分に効果的であると言える．一方で，FALSE-STALL-XACT の削減も認められたが，そもそも総実行サイクル数に占める割合が小さいため，削減されたことによる影響は少ない．

次に，総実行サイクル数に着目する．既存モデル (B) と比較した場合，一部性能が変化していないものもあるが，概ね性能が向上している．特に，FALSE-STALL-NONXACT の占める割合が非常に大きい Sortedlist では，提案手法により FALSE-STALL-NONXACT を解消したことで大幅にサイクル数が削減された．ここで，3 つのプログラムにおける，既存モデル (B) と比較した場合のキャッシュミスペナルティサイクルの増大率を**表 9** に示す．表より，各プログラムで発生するミスペナルティサイクル数は既存モデル (B) と比べてあまり変化がなく，表 7 で示した提案モデル (P) に

表 9: 提案モデル (T_{16}) におけるキャッシュミスペナルティサイクル数の増大率

bench	/thrs	L1 penalty cycles	L2 penalty cycles
Vacation	/4	1.5%	-2.3%
	/8	-2.9%	-3.6%
Prioque	/4	0.1%	1.8%
	/8	3.9%	-4.5%

おけるミスペナルティサイクル数と比べても，キャッシュミスによる悪影響は少ない．したがって，提案モデル (T_{16}) ではキャッシュミスの増大を抑えつつ false-stall の削減を達成できると言える．

このように，本評価結果では，特に Sortedlist に関して実行サイクル数の大幅な削減が認められる．ここで，既存モデル (B) を適用した場合の Prioque 及び Sortedlist に関しては，6.1 節で説明したように，メモリ配置状況によってはプログラム内で発生する false-stall が非常に多くの割合を占める可能性がある．このような不利な状況は，記憶領域をある一定のブロックサイズで切り出して扱うような環境下ではどんなプログラム上でも顕在化する可能性があり，その予測は困難である．さらに，不利な状況を防止するためには，コンパイル後の変数のメモリ配置状況を考慮してプログラムを設計する必要があるが，そのような作業はプログラマにとって負担となりうる．しかし，提案モデル (T_{16}) ではこのような不利な状況によって生じる false-stall を隠蔽することができる．

ここで，提案モデル (T_{16}) で必要となる R/W テーブルのハードウェアコストを見積もる．まず，R/W テーブルには，競合が発生したキャッシュラインの数だけのエントリが必要となる．そこで，各プログラムにおいて，R/W テーブルひとつ当たりに使用されたエントリ数を**表 10** に示す．表より，最大で 17 エントリあれば，全てのプログラムにおいて R/W テーブルが溢れることなく false-stall を防止することができる．ここで，R/W テーブルのひとつのエントリ当たりに必要となる Tag は，アドレス幅 $W = 64$ bit，キャッシュラインサイズ $L = 64$B である場合，$W - \log_2 L = 64 - \log_2 64 = 58$ bit であり，また R，W^{my} 及び W^{other} はそれぞれ 16 bit である．したがって，ひとつの R/W テーブルは幅 106 bit 深さ 17 行の RAM で構成できるため，32 スレッドを実行可能な 32 コア構成のプロセッサの場合では R/W テーブルサイズの総和は約 7kB と少量である．

提案モデル (T_{16}) で必要となる R/W テーブルのハードウェアコストは， 5.3 節で説

表 10: R/W テーブルの使用エントリ数

bench	/thrs	最大	平均
Vacation	/4	15	13.0
	/8	17	13.0
Prioque	/4	9	7.6
	/8	11	8.7
Sortedlist	/4	3	3.0
	/8	3	3.0

明したように，提案モデル (T_{16}) を提案モデル (P) と融合させたモデルを適用することで削減できると考えられる．これは，融合モデルにより，表 10 に示した R/W テーブルに必要なエントリ数の削減が期待できるからである．しかし，前述したように，Genome を除く 3 つのプログラムにおける，false-stall の削減に必要な R/W テーブルのサイズは高々 7kB と少ない．さらに，提案モデル (P) の評価結果では，FALSE-STALL-NONXACT の削減は達成できたが，キャッシュミスの増大により著しく実行サイクル数が増大した．以上の結果から，融合モデルを適用した場合の効果は少ないと考えられるため，これらのモデルを融合することによるメリットは，現状では見いだせない．

提案モデル (T_4)

評価の結果，3 つのプログラムにおいて FALSE-STALL-NONXACT が提案モデル (T_{16}) と同様に削減されたことが確認できる．さらに，総実行サイクル数は，Vacation 及び Sortedlist では提案モデル (T_{16}) とほぼ同じ結果が得られた．しかし，提案モデル (T_4) では，キャッシュラインを 4 等分したのちのひとつのブロックを，競合を検査する最小単位として扱うため，提案モデル (T_{16}) と比べて false-stall の発生する可能性が高まると考えられる．そこで，R/W テーブル上で管理される変数の配置されるアドレスを詳細に調査したところ，3 つのベンチマークにおいて FALSE-STALL-NONXACT が発生する異なるデータ構造は，同一キャッシュライン上のある程度離れた位置に配置されていることがわかった．例えば，Sortedlist では 3.2 節で説明したように，ライブラリ関数 srandom() によって確保される変数と，リスト構造が同一キャッシュライン上に存在しており，前者の末尾部分がはキャッシュラインの前半，後者の先頭部分がラインの後半の領域にそれぞれ配置されていた．したがって，提案モデル (T_4) を適用した場合に，競合の検査を行う，キャッシュラインの 1/4 のサイズのブロック内には，異なるデータ構造が存在しないため，結果として提案モデル (T_{16}) と同等に FALSE-STALL-NONXACT

を削減することができた．残りのプログラムでも，FALSE-STALL-NONXACT の発生するようなデータ構造はある程度の大きさを有しているため，キャッシュラインサイズの 1/4 程度の大きさのブロックを最小単位として競合を検査するだけで，提案モデル (T_{16}) と同等の効果を得ることができた．この結果より，少なくとも今回評価したプログラムに関しては，提案モデル (T_{16}) ほどの細かい単位で競合を検査する必要がないことがわかった．

しかし，FALSE-STALL-XACT に関しては，配列の要素や構造体のメンバ変数等，競合の検査する対象が FALSE-STALL-NONXACT と比べて小さいため，検査する最小単位をより細かくする必要があると考えられる．しかしながら，FALSE-STALL-XACT が総実行サイクル数に占める割合は非常に小さいため，競合を検査する単位を大きくしたことで FALSE-STALL-XACT が増大したとしても，その影響は少ない．さらに，配列や構造体を持つプログラムは，トランザクション内でそれらの要素のなかでもただひとつのみにアクセスする場合は少なく，複数の要素にアクセスすることで同一の要素における競合が発生すると考えられるため，FALSE-STALL-XACT を引き起こすような状況は少ないと想定される．したがって，より大きな影響を与える可能性のある FALSE-STALL-NONXACT を削減しつつ，必要なハードウェア量も抑えることができる提案モデル (T_4) は，提案モデル (T_{16}) よりも適切なモデルであると言える．

ここで，提案モデル (T_4) で必要となる R/W テーブルのハードウェアコストを見積もる．R/W テーブルのひとつのエントリ当たりに必要な R，W^{my} 及び W^{other} はそれぞれ 4 bit である．また，競合の発生する変数の存在するキャッシュラインのアドレスは，競合を検査する単位を小さくしても変化しないと考えられるため，ひとつの R/W テーブルに必要な RAM の行数は提案モデル (T_{16}) と同じであるとする．したがって，ひとつの R/W テーブルの構成には幅 70 bit，深さ 17 行の RAM が必要となるため，R/W テーブルサイズの総和は約 5kB と，ごく小さいものとなる．したがって，提案モデル (T_{16}) と比べて約 2/3 の大きさのハードウェアコストで，提案モデル (T_{16}) と同等の効果が得られる．

また，Prioque を 8 スレッドで実行した場合では，提案モデル (T_{16}) と比べて主に XACT が減少したことにより，総実行サイクル数が減少していることがわかる．しかし，Prioque を詳細に調査した結果，試行ごとに発生するアボート回数が大きく変動することがわかった．例えば，既存モデル (B) において Prioque を 8 スレッドで実行した場合，各試行における総アボート回数は最大で 2663 回，最小では 830 回であった．したがって，提案モデル (T_4) の評価時にはアボート回数が少ない場合が重なったため，

提案モデル (T_{16}) と比べて XACT が減少したと考えられる．しかし，先述したとおり，FALSE-STALL-NONXACT は削減されているため，R/W テーブルによる競合検出手法の効果は十分得られていると言える．

提案モデル (T_2)

Prioque 及び Sortedlist では，提案モデル (T_{16}) 及び (T_4) とほぼ同量の FALSE-STALL-NONXACT が削減され，総実行サイクル数もこれらの提案モデルと同等の結果を得ることができた．これは，Sortedlist 及び Prioque において FALSE-STALL-NONXACT の発生する 2 つの異なるデータ構造は，そのサイズがある程度大きく，それら 2 つのデータ構造が同一キャッシュライン上の前半及び後半部分にそれぞれ配置されたため，提案モデル (T_{16}) 及び (T_4) と同等に FALSE-STALL-NONXACT を削減する効果が得られたと考えられる．

一方で，Vacation では，提案モデル (T_{16}) 及び (T_4) ほど FALSE-STALL-NONXACT が削減されなかった．これは，Vacation において FALSE-STALL-NONXACT の発生するデータ構造は，Prioque 及び Sortedlist と比べてそのサイズが小さく，キャッシュラインサイズの半分の大きさの領域内に複数の異なるデータ構造が配置される可能性があるため，競合検出の単位をキャッシュラインの 1/2 のサイズのブロックとしても無駄な競合を解消することができないからである．したがって，Vacation において FALSE-STALL-NONXACT を解消するためには，競合検出の単位をキャッシュラインサイズの 2 等分した領域よりもさらに細分化する必要がある．

ここで，提案モデル (T_2) で必要となる R/W テーブルのハードウェアコストを見積もる．ひとつの R/W テーブルエントリ当たりに必要な R，W^{my} 及び W^{other} はそれぞれ 2 bit であり，ひとつの R/W テーブルに必要な RAM の行数は提案モデル (T_{16}) と同じであるとする．したがって，ひとつの R/W テーブルの構成には幅 64 bit，深さ 17 行の RAM が必要となるため，R/W テーブルサイズの総和は約 4kB となる．したがって，提案モデル (T_{16}) と比べて約 1/3，提案モデル (T_4) と比べて約 4/5 の大きさのハードウェアコストとなる．このように，提案モデル (T_2) では，提案モデル (T_{16}) 及び (T_4) と比べてハードウェアコストが少なく済むが，プログラムによっては提案モデル (T_{16}) 及び (T_4) ほどの効果を得ることができない．

7　関連研究

ひとつのキャッシュライン上に存在する異なる変数に対して複数のスレッドがそれぞれアクセスする場合，無駄な競合が発生することを説明してきた．一方で，マルチ

コア環境において，複数のスレッドが単一キャッシュラインを共有する状況では false-sharing が引き起こされることが知られている [11]．この現象を解決するために、各プロセッサ・コアごとにプライベートなヒープ領域を保持・管理することで，スレッドローカルに定義された変数が同一キャッシュライン上に配置されることを防ぐ手法である LKmalloc[12] や Hoard[13] が提案されている．これらの手法を LogTM にも応用することで，スレッドローカルに定義された異なる変数や，グローバル変数とスレッドローカル変数との間で発生する無駄な競合を防止することができる．

また， 3.1 節で挙げた，個々の変数に対する競合を検出できないという問題点以外に着目することで，LogTM を高速化する手法が多く研究されている．例えば，トランザクションがアボートされた場合に，そのトランザクションを途中から再実行させることで，再実行に必要な命令数を削減する部分ロールバック手法 [14, 15] が提案されている．これらの手法では，再開可能な位置 (チェックポイント，CP) をプログラマが記述する必要があり，さらに，ひとつのトランザクション内で設定可能な CP の数に制限があるという問題があった．

この問題を解決するために，伊藤ら [16] は頻繁に競合が検出されるロード・ストア命令の直前に自動的に CP を設定する手法を提案している．同時に，CP の設定数の制限を緩和する手法も提案している．しかし， 2.2.1 項で説明した possible_cycle フラグを用いるモデルではなく，競合発生時に即座にトランザクションをアボートさせるような，よりアボートが発生しやすいモデルをベースに評価している．加えて，評価に対する考察が少なく，提案モデルのベースとしている Williullah らによる手法 [15] 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．

また，適切な並列実行トランザクション数を動的に制御する手法 [17] が提案されている．この手法では，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，同時実行可能なトランザクション数を動的に制限することでアボートを防止し，アボートに要するサイクル数を削減している．しかし，この手法を適用した場合のオーバーヘッドや，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ひとつのみであり，効果の汎用性も明らかではない．

さて，複数のスレッドがひとつのキャッシュライン上に存在する異なるデータにアクセスする可能性があることは，記憶領域をブロック単位で扱うようなマルチコア・アーキテクチャが孕む本質的な問題である．この問題は，LogTM をはじめとする HTM において，false-stall が発生するという影響を及ぼす危険性がある．false-sharing の解決に関する研究を HTM に応用することで，false-stall の一部を解決できるが，グローバ

ルに定義されたデータ構造が同一ライン上に存在する場合や配列の要素に対する無駄な競合を防止することはできない。また，部分ロールバックや同時実行スレッド数の動的制御に関する研究では，そもそも false-stall に関しては全く考慮されていない．したがって，本研究では競合検出の単位に着目することで，マルチコア・アーキテクチャが持つ本質的な問題によって顕在化する影響を軽減する．

8 おわりに

本研究では，既存の LogTM によってキャッシュライン単位で行っていた競合の検出操作を細粒度化する 2 つの手法により，同一キャッシュライン上の異なる変数に対して複数のトランザクションがアクセスすることで発生する，本来ならば検出する必要のない無駄な競合を防止した．1 つ目の手法では，このような検出される必要のない競合が発生する可能性のある変数を自動的に異なるキャッシュラインに分散配置させ，2 つ目の手法では，各変数へのアクセス情報を記憶する，R/W テーブルというハードウェアを保持することで，変数単位で競合を検査可能とした．

STAMP ベンチマーク及び GEMS 付属のマイクロベンチマークを用いてシミュレーション評価した結果，提案手法により前述した無駄な競合の発生が防止できることを確認した．また，既存の LogTM に比べて，1 つ目の提案手法では最大 83.9%の実行サイクル数が削減できたが，平均では 7.0%の悪化となり，2 つ目の提案手法では，最大 82.9% ，平均 28.0%の実行サイクル数の削減が確認できた．

今後の課題として，以下の 3 つが考えられる．まず，簡易的な実装を施した 1 つ目の提案手法に対して，メモリ領域を効率的に利用する手法を実装することが挙げられる．今回実装した提案手法 1 では，分散配置した変数の間に無駄なメモリ空間が存在してしまうため，そのような空間に対して競合に関係しない変数を挿入することで，キャッシュミスの増大を防ぐことができると考えられる．

次に，2 つ目の提案手法で用いた R/W テーブルの利用効率を向上させることである．R/W テーブルには過去に一度でも競合が発生したキャッシュラインに対応したエントリが挿入されるが，以降一度もそのラインで競合が発生しない場合，エントリが R/W テーブルに残り続ける．このような今後参照されることのない無駄なエントリが増大することで R/W テーブルがオーバーフローする可能性があるが，今回の提案した手法では，トランザクションをアボートさせることで無駄なエントリを含む，全てのエントリを削除していた．そこで，そのような無駄なエントリの存在を判別し，定期的に R/W テーブルから削除する機構を追加することで，R/W テーブルの利用効率を向

上させることができると考えられる．

最後に，部分ロールバック手法との連携がある．トランザクションをその途中から再実行する部分ロールバック手法と組み合わせることで，更なる高速化を実現できると考えられる．

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，終止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江藤正道氏と堀場匠一朗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. Hiroki ASAI, Tomoaki TSUMURA, Hiroshi MATSUO: “Proposition of Criteria for Aborting Transaction based on Log Data Size in LogTM”, Proc. 1st Int’l. Conf. on Networking and Computing (ICNC’10), Higashi-Hiroshima, Japan, pp.95–103 (Nov. 2010)

報文

1. 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, (to appear) (Mar. 2012)
2. 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)
3. 浅井 宏樹, 津邑 公暁, 松尾 啓志: “ログエントリ数を考慮したLogTM アボート対象選択手法とその評価”, 情処研報 (SWoPP2010), Vol.2010-ARC-190, No.4, pp.1–9

(Aug. 2010)

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] 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).

[8] 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).

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

[10] 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).

[11] Torrellas, J., Lam, M. S. and Hennessy, J. L.: False sharing and spatial locality in multiprocessor caches, *IEEE Transactions on Computers*, Vol. 43, pp. 651–663 (1994).

[12] Larson, P.-A. and Krishnan, M.: Memory allocation for long-running server applications, *Proc. of the 1st international symposium on Memory management*, ISMM '98, New York, NY, USA, ACM, pp. 176–185 (1998).

[13] Berger, E. D., McKinley, K. S., Blumofe, R. D. and Wilson, P. R.: Hoard: a scalable memory allocator for multithreaded applications, *SIGPLAN Not.*, Vol. 35, pp. 117–128 (2000).

[14] 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).

[15] 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).

[16] 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ポジウム SACSIS2011 論文集, pp. 324–331 (2011).

[17] 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).