卒業研究論文

ログエントリ数を考慮した LogTMのアボートコスト削減手法

指導教員　　松尾 啓志 教授
　　津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成 18 年度入学　18115008 番

浅井 宏樹

平成 22 年 2 月 8 日

ログエントリ数を考慮したLogTMのアボートコスト削減手法

浅井 宏樹

内容梗概

マルチコア環境におけるTLPを活用した並列プログラミングでは，共有リソースのアクセスにロックが多く用いられている．しかしロックを用いた場合，デッドロックや並列性の低下の問題点がある．そこでロックを用いない同期制御機構としてLogTMが提案されている．

LogTMはトランザクションを投機的に実行する．トランザクションのアトミシティを保つために，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競合を検出する．競合が発生した場合はトランザクションの実行を停止してそれまでの結果を全て破棄する．この操作をアボートという．アボートしたトランザクションは開始時点まで戻り，再実行する．

しかし，LogTMの問題点としてアボートコストが高いことが挙げられる．LogTMはアボートの対象をトランザクションの開始時刻で決定しているため，アボートコストが高いトランザクションをアボートしてしまう可能性がある．したがって競合が頻繁に起きるようなプログラムでは性能が低下する恐れがある．

本稿では，ログエントリ数を使用してアボートコストのより低いトランザクションをアボートさせる手法を提案する．これによりアボートコストの高いトランザクションのアボートを防ぐことができ，プログラム全体のアボートコストを削減することができる．さらに，ログエントリ数のみでなくトランザクションの開始時刻も考慮したハイブリッド型モデルを提案する．これにより，ログエントリ数を考慮したアボートが有効でない場合でも性能の向上が期待できる．

提案手法の有効性を検証するため，既存のLogTMを拡張して提案手法を実装し，シミュレーション評価を行った．評価の対象としてSPLASH-2ベンチマークのうち，BarnesとRaytraceを選択した．その結果，BarnesとRaytraceの両方で既存のLogTMに比べて最大で約4%，平均で約2%の実行サイクル数が削減でき，提案手法の有効性が確認できた．

ログエントリ数を考慮したLogTMのアボートコスト削減手法

目次

1 はじめに

今日までに，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 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．この流れを受け，CPU がマルチコア化されるとともに，スレッドレベル並列性 (Thread-Level Parallelism:TLP) が注目されている．

マルチコア環境における TLP を活用した並列プログラミングに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並列プログラミングを用いることが多い．また，共有メモリ型並列プログラミングでは共有リソースに対して排他制御を行う必要がある．そのような同期制御機構として一般的にはロックが多く用いられている．しかし，ロックを用いた場合はデッドロックや並列性の低下の問題点がある．さらにプログラマはロックの粒度を考慮したプログラムを構築する必要があり，ロックの利用が困難である一因となっている．そこで，ロックを用いない同期制御機構として LogTM[1] が提案されている．

LogTM はロック・フリーな機構であるトランザクショナル・メモリをハードウェア上に実装したハードウェア・トランザクショナル・メモリのひとつである．LogTM ではトランザクションを投機的に実行する．トランザクションとはクリティカルセクションを含む一連の命令列のことを指す．投機実行するトランザクションのアトミシティを保つために，トランザクション内で発生したメモリアクセスと他のトランザクションで発生したメモリアクセスとが競合しているかどうかを検査する．競合が発生した場合は，トランザクションの実行を停止してそれまでの結果を全て破棄する．この操作をアボートという．アボートされたトランザクションはそのトランザクションが開始した時点まで戻って再実行する．このようにして共有リソースの同期を実現している．

しかし，LogTM の問題点としてアボートコストが高いことが挙げられる．LogTM では共有リソースへのストアアクセス時にストアで書き換えられる前の値をログと呼ばれる領域に退避させ，競合が発生した場合にログに退避された値を元のメモリアドレスに書き戻す操作を行う．つまりログに退避された情報が多ければアボート時の書き戻しコストが大きくなる．したがって競合が頻繁に起きるようなプログラムでは性能が低下する可能性がある．

このため本研究ではログエントリ数のより少ないトランザクションを動的に選択してアボートさせる手法を提案する．これにより，ログからの書き戻しコストの高いト

ランザクションのアボートを防ぐことができ，結果としてプログラム全体のアボートコストが削減できる．さらに，ログエントリ数に加え，既存の手法で用いられているトランザクションの開始時刻を考慮したハイブリッド型モデルを提案する．これにより，ログエントリ数によるアボートが有効でない場合でも性能の向上が期待できる．

以下，2 章では本研究の背景としてトランザクショナル・メモリ及び LogTM の概要や，その実装について説明する．3 章では LogTM の欠点を述べるとともに本研究が提案するモデルについて説明し，4 章ではその実装方法について説明する．5 章で本手法を評価し，6 章で結論を述べる．

2 研究背景

本章では，本研究で取り扱うトランザクショナル・メモリ及び LogTM について述べる．

2.1 トランザクショナル・メモリ

マルチコア・プロセッサにおける並列プログラミングでは，共有メモリ型のプログラムが多く用いられる．共有メモリ型の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ので，共有リソースのアクセスに対して排他制御を行う必要がある．そのような同期制御機構として一般的にはロックが多く用いられている．しかし，ロックを用いた同期にはデッドロックや不要なロックによるオーバヘッド，並列に実行するスレッド数の増加に伴う性能低下などの問題を抱えている．またプログラムごとに最適な粒度を設定するのは難しく，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同期制御機構として**トランザクショナル・メモリ**[2] が提案されている．

トランザクショナル・メモリはメモリアクセスに対してデータベースのトランザクション処理を適用した同期手法である．トランザクショナル・メモリでは，クリティカルセクションを含む一連の命令列をトランザクションと定義する．Herlihy らによって，トランザクションは以下の性質を満たすと定義されている．

シリアライザビリティ (直列可能性):
並行して実行されたトランザクションの実行結果は，同一の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全く実行されないかのいずれ

かである．

以上の性質を保証するために，トランザクショナル・メモリはあるトランザクションが他のトランザクションと同じメモリアドレスにアクセスするかどうかを検査する．つまり，自分以外のトランザクションからアクセスされたメモリアドレスと，自身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，トランザクションの性質を満足しないため競合として検出する．これを競合検出という．競合が発生した場合，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．アボートしたトランザクションはそのトランザクションの開始時点まで戻り，再実行する．一方でトランザクションの終了までに競合が検出されなかった場合，トランザクション内で実行された結果を全てメモリに反映させる．これを**コミット**という．

このようにトランザクショナル・メモリを用いることでトランザクションは競合しない限り並列に実行することができる．これによりロックよりもプログラムの並列性が上がるため，実行速度が向上する．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，ロックよりも容易に扱うことができる．

Herlihy らはトランザクショナル・メモリをハードウェア上に実装する手法を提案している．この手法では命令セットを拡張し，トランザクション内でのメモリアクセスと競合検出，及びコミットをサポートする．また，トランザクション内でストア命令を実行したとき，キャッシュ上の値は更新されるが，主記憶上の値はそのまま維持される．アボート時にはキャッシュ上の値を破棄することでトランザクションを開始時点の状態まで戻して再実行することができる．

2.2 LogTM

次にトランザクショナル・メモリのひとつの実装方式である LogTM に導入された 2 種類の概念とその実装方法及びハードウェア構成について説明する．

2.2.1 データのバージョン管理

LogTM はトランザクションを投機的に実行する．投機的実行では実行結果が破棄される可能性があるので，トランザクションは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．これをバージョン管理という．そこで LogTM では**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ログに保存する．一方でストア命令の結果はストア対象のメモリアドレスに書き込まれる．これによりアクセスしたデータ

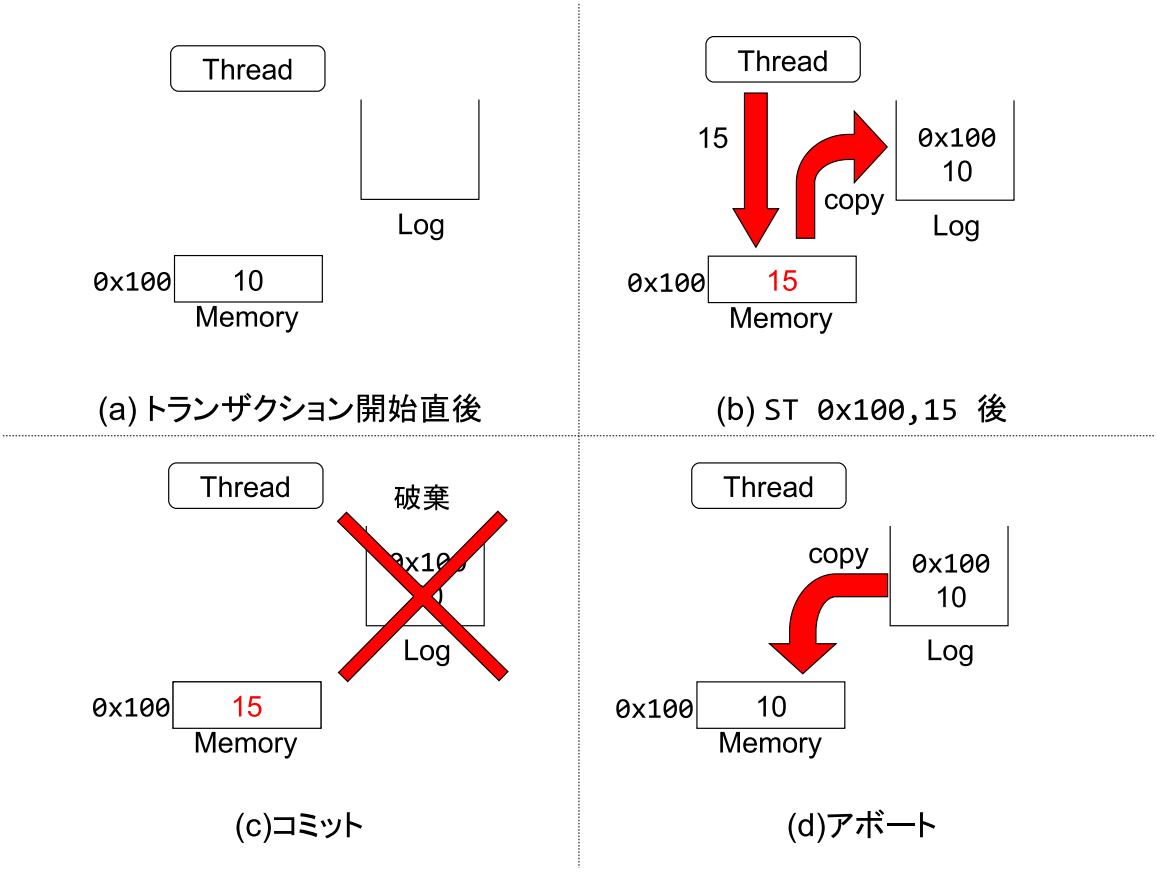

図 1: データのバージョン管理

の古いバージョンを管理することができる．

また，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，ログに保存されるのは最初の 1 回だけとする．なぜなら，トランザクションの再実行にはトランザクション開始時点でのメモリ状態を知るだけで十分だからである．

図 1 はデータのバージョンが管理される様子を示している．図 1 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しており，図 1 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．ここでトランザクションの実行を開始したとする．トランザクションの開始時点ではメモリの 0x100 番地には 10 が格納されており，ログが割り当てられる (図 1(a))．その後 `ST 0x100,15` が実行されたとすると，先ほど述べたようにストアアクセスしたメモリアドレスである 0x100 番地と書き換えられる前の値である 10 が主記憶からログに保存され，ストアの結果である 15 が主記憶に上書きされる (図 1(b))．

次に投機的実行が成功した場合を考える．投機的実行が成功した場合はトランザク

ション内で変更したメモリ状態を全て主記憶に反映させるコミット操作を行う．しかし，全てのストア結果は既に主記憶に保存されているため，メモリアクセスを行わなくてもメモリ状態は反映されている．したがってログの内容を破棄する操作のみを行う (図 1(c))．

一方で投機的実行が失敗した場合を考える．投機的実行が失敗した場合はトランザクションの内部で変更があった値を全て破棄し，トランザクション開始時点のメモリ状態まで戻すアボート操作を行う．アボートによってログに保存された全ての値を元のメモリアドレスに書き戻し，トランザクションの実行中に変更があった値を全て破棄する (図 1(d))．これによりトランザクション開始時点のメモリ状態まで戻すことができる．

ここでコミット操作とアボート操作の違いについて説明する．先ほど述べたようにコミット操作ではメモリアクセスが全く行われず，反対にアボート操作ではログに保存された値を全て主記憶に書き戻すためログサイズに比例してメモリアクセスが増加する．これはコミットがトランザクションの終了時に必ず行われる操作であることを考慮するためである．つまり必ず行われる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ている．

また，アボート時にはメモリ状態と同様にレジスタの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．LogTM ではトランザクションの開始時に開始時点のレジスタ状態を保存することで，アボート時にレジスタ状態を回復させている．

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，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しているかどうかを検出する必要がある．この操作を競合検出という．

競合の検査を行うためには，どのメモリアドレスがトランザクション内部で投機的にアクセスされたかを管理することが必要である．そのためキャッシュライン上に新しく read ビット及び write ビットを追加する．read ビットはトランザクション内でリードアクセスが発生した場合にセットされ，write ビットはライトアクセスが発生した場合にセットされ，トランザクションのコミット及びアボート時にリセットされる．さらに，競合を検知した場合にはトランザクションに競合を通知する必要がある．そのためキャッシュの一貫性を保つキャッシュ・コヒーレンス・プロトコルを拡張する．LogTM では一貫性のモデルにディレクトリベース [3] の Illinois プロトコル [4] を採用している．例えばあるメモリアドレスにアクセスする場合，キャッシュの一貫性を保つためキャッ

シュラインの状態を変更させる要求が送信される．これをキャッシュ・コヒーレンス・リクエストという (以下，リクエストと省略する)．拡張したキャッシュ・コヒーレンス・プロトコルはリクエストによって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．これによりトランザクション内でアクセスしようとしたメモリアドレスが他のトランザクションによって既にアクセスされていることを検出することができる．具体的には以下の 3 パターンのアクセスが行われた場合に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たパターンである．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が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ため，アトミシティを満たさない．

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たパターンである．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が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ため，アトミシティを満たさない．

write after write:

write ビットがセットされたアドレスが，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が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ため，アトミシティを満たさない．

以上のような競合パターンが検出されると，競合を検出したトランザクションからリクエストを送信したトランザクションに対して nack が送信される．nack を受信したトランザクション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．ストールしたトランザクションは同じアドレスに対するリクエストを送信しつづけることで，競合したトラン

ザクションの終了を検知する．一方で競合が発生しなかった場合は ack が送信される．

トランザクションが並列に実行して競合を検査する動作モデルを**図 2** に示す．図 2 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 2 中にあるメモリアドレスは全て共有リソース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説明する (図 2(a))．図 2(a) では最初に T1 が 0x100 番地に対するロード命令を実行する (図 2 t1)．このとき T1 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．図 2 t1 の時点では 0x100 番地へのアクセスは行われていないため，T1 はロード命令を実行することができる．その後 T2 が 0x100 番地に対してロード命令を実行する．このとき T2 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 (図 2 t2)．図 2 中の `request` は送信されたリクエストを示している．実際にはキャッシュの状態を一括で管理しているディレクトリから送信されているが，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．リクエストによって T1 が既に 0x100 番地にアクセスしていることがわかるが，先ほど説明した競合パターンに当てはまらないので競合は発生しない．したがって T2 は ack を受信し，命令を実行することができる．

次に競合が発生する場合について説明する (図 2(b))．図 2(a) と同様に T1 は 0x100 番地に対するロード命令を実行する．その後 T2 が 0x100 番地に対してストア命令を実行する (図 2 t3)．このとき T2 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が，このリクエスト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，競合が検出される．よって T2 からのリクエストを受信した T1 は競合したことを通知するために T2 へ nack を送信する (図 2 t4)．その後 nack を受信した T2 はストールする (図 2 t5)．トランザクションの実行が進み，T1 がコミットする (図 2 t6) と，T2 は 0x100 番地にアクセスできるようになったことをストール中に再送したリクエスト (図 2 request2,request3) により知ることができる．したがって ack を受信した T2 はストール状態から復帰し，トランザクションの実行を続けることができる (図 2 t7)．

しかし，競合によりストールしたトランザクションが複数発生するとそれらがデッドロック状態に陥る危険性がある．例えばトランザクション T1 と T2 が投機的に実行していると仮定する．今，T1 から T2 に nack を送信し，その後 T2 から T1 に nack を送信したとする．このときお互いがお互いの終了を待ってしまうためデッドロックに

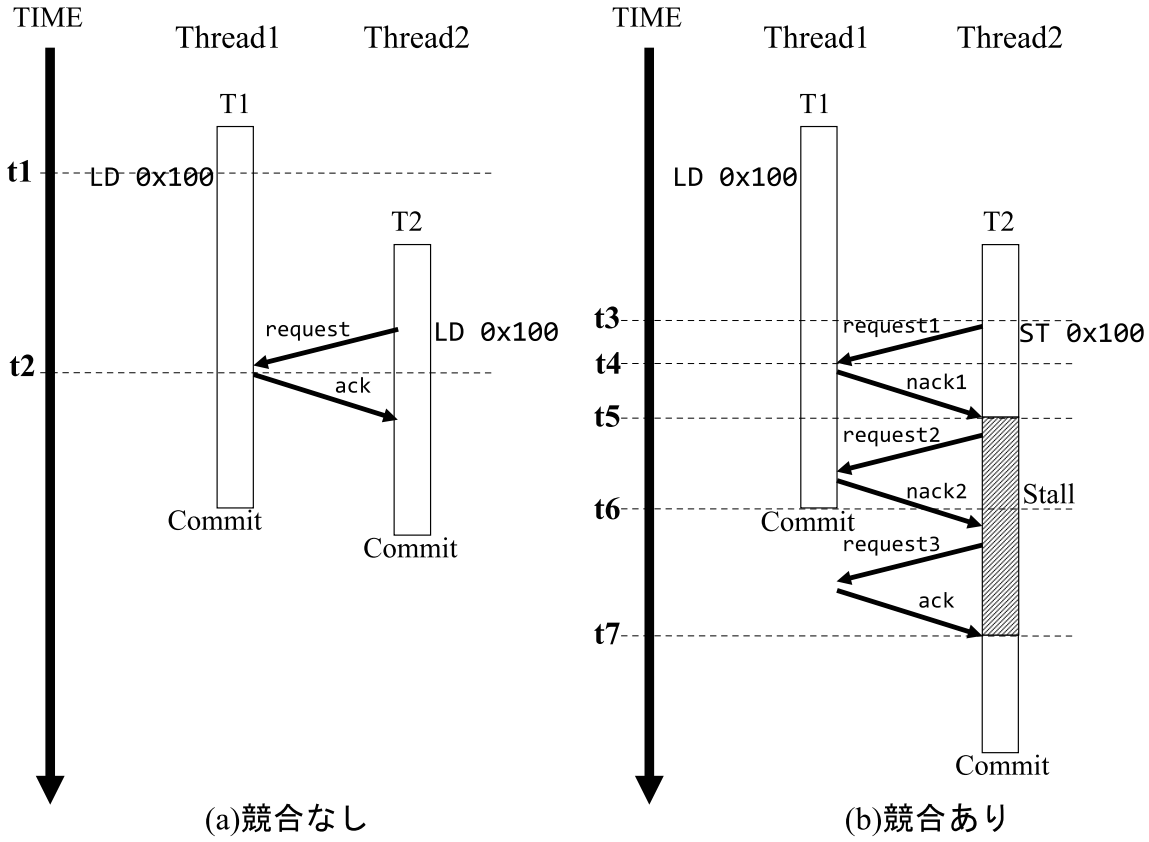

図 2: 競合の検査

陥る．このようなデッドロック状態を解消するために，デッドロックに陥ったトランザクションのどちらかをアボートさせる．LogTM では開始時刻がより遅いトランザクションをアボートの対象として選択している．なぜなら，開始時刻が早いトランザクションはより多くのメモリアクセスを行っている可能性が高いため，競合の頻発を防ぐために早くコミットすることが望ましいからである．

デッドロック時に開始時刻の遅いトランザクションをアボートさせるためには，トランザクションの開始時刻を比較する必要がある．そのため LogTM では各スレッドがトランザクション開始時のタイムスタンプを保有している．また，デッドロックしたことを検知するためには，トランザクションが他のトランザクションとの競合を検出してストールさせていることを知る必要がある．そのため各スレッドは possible_cycle フラグを保有する．possible_cycle フラグは TLR's distributed timestamp method[5] で用いられている．この手法では，あるトランザクションがより早く開始したトランザクションに対して nack を送信したとき，そのトランザクション対して possible_cycle フラ

グがセットされる．そして possible_cycle フラグがセットされているトランザクションが，自身よりも早く開始したトランザクションから nack を受信した場合，デッドロックが発生したとみなしてアボートする．

デッドロックが発生してアボートする場合の動作モデルを**図 3** に示す．Thread1 では例 1 にあるトランザクション T1 が実行され，Thread2 では例 2 にあるトランザクション T2 が実行されるとする．まず，T1 が実行を開始した後に T2 が実行を開始する．次に T1 で `ST 0x100` を実行し，その後に T2 で `ST 0x200` が実行される．さらにその後 T1 で 0x200 番地に対するロードが実行されると，T1 はリクエストを送信する (図 3 t1)．リクエストを受信した T2 は競合したことを検知するため T1 へ nack を送信する．このとき T2 は自分よりも早く開始したトランザクションである T1 へ nack を送信したので，T2 の possible_cycle フラグがセットされる (図 3 t2)．T2 から送信された nack を受信した T1 はストールする (図 3 t3)．その後，T2 で `ST 0x300`，`ST 0x400` が実行されるが，T1 と競合しないため処理が進む．さらにその後 0x100 番地に対するロードが実行されると (図 3 t4)，T1 と競合するため T2 は nack を受信する (図 3 t5)．このとき，T2 は自身よりも早く開始したトランザクションである T1 から nack を受信し，かつ T2 には possible_cycle フラグがセットされているため，T2 はアボートする．もし T2 がアボートしなければ，T1 は T2 が終了するまでストールし，同時に T2 は T1 が終了するまでストールするためデッドロック状態に陥ってしまう．したがってこの時点でアボートすることが十分であるといえる．

アボートした T2 は開始時点まで戻り，トランザクションを再実行する (図 3 t6)．また，T2 がアボートしたことにより T1 は 0x200 番地にアクセスできるようになるため，T2 のストール状態が解消される (図 3 t7)．

2.3 LogTM のハードウェア構成

LogTM のハードウェア構成を**図 4** に示す．図 4 に示すように，プロセッサコアごとに 1 次データキャッシュ，2 次データキャッシュ，キャッシュコントローラを 1 つ持ち，主記憶は全てのコアによって共有されている．1 次及び 2 次データキャッシュには，2.2.2 項で述べたように read ビット及び write ビットが追加されている．また，キャッシュコントローラではトランザクションの開始時点のタイムスタンプが保持され，Possible_cycle ビットと Overflow ビットが実装されている．Possible_cycle ビットは possible_cycle フラグを管理し，Overflow ビットは overflow フラグを管理する．overflow フラグは，トランザクション実行中にキャッシュがオーバーフローした場合にセットされる．

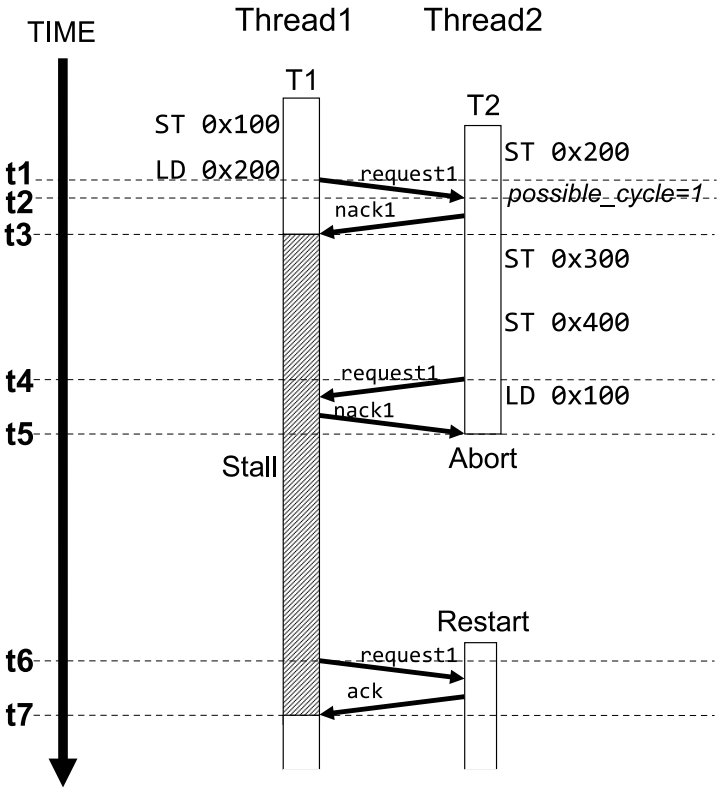

図 3: 既存の LogTM でのアボート対象の選択

例 1：トランザクション T1 で実行される命令

```
1:  ST 0x100
2:  LD 0x200
```

例 2：トランザクション T2 で実行される命令

```
1:  ST 0x200
2:  ST 0x300
3:  ST 0x400
4:  LD 0x100
```

3　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．

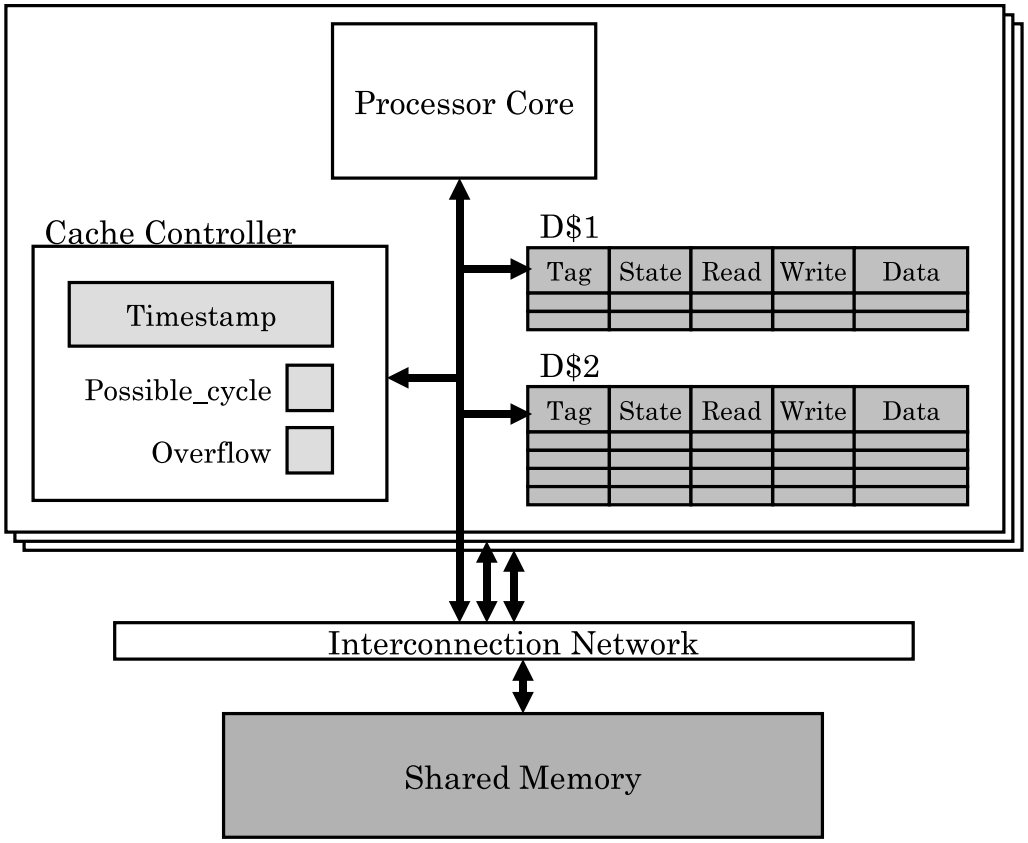

図 4: 既存 LogTM の構成

3.1 LogTM の問題点

既存モデルである LogTM の問題点としてアボートコストが高いことが挙げられる．アボートコストとは，アボート時にメモリ状態を開始時点まで戻すためにログに保存された値を主記憶に書き戻すときの書き戻しコストである．よって書き戻しコストはログに保存されたエントリ数に比例する．ここでログに保存されたエントリの総数をログエントリ数と呼ぶ．ログエントリ数はトランザクション内でストア命令が発行されるたびに増加する．一般に主記憶アクセスはアクセスレイテンシが大きいため，ログに保存されたエントリ数が多ければメモリアクセスに多大なコストがかかる．したがって競合が頻繁に発生するようなプログラムが実行されるとアボートが頻発して性能が低下する可能性が高い．ただし，2.2.1 項で述べたように同じメモリアドレスに対するエントリは重複して保存されないため，異なるメモリアドレスに対して多くストアを実行するとアボートコストが増大するといえる．

しかし，既存の LogTM はアボートコストがどれだけかかるかを全く考慮せずにアボートを実行してしまう．これはアボート対象の選択にタイムスタンプが用いられる

ためである．したがってアボートコストのより大きなトランザクションがアボートされる可能性がある．そこで，本研究ではアボートコストがより少ないトランザクションをアボートさせる手法を提案する．これによってプログラム全体の実行時間を削減することができる．

3.2 ログエントリ数を用いたアボート対象の選択方法

本研究ではアボートコストがより少ないトランザクションを選択してアボートさせることで，アボートコストを最小限に留める手法を提案する．アボートコストのより少ないトランザクションをアボートさせるためには，トランザクションのアボートコストを比較する必要がある．アボートコストは前項で述べたようにログに保存されたエントリ数に比例するため，トランザクションが保有するログのエントリ数がより少ないトランザクションをアボートさせることで，アボートコストのより少ないトランザクションをアボートさせることができる．

まず，既存の LogTM ではアボートコストが高くなることを説明する．2.2.2 項の図 3 で説明した例ではトランザクション T2 がアボートしている．このとき，T2 が保有するログには実行されたストア命令の回数だけエントリが追加されている．したがって T2 では 3 箇所のメモリアドレスに対してストアを実行したため，アボートには 3 箇所のメモリアドレスへの書き戻しが必要になる．一方で T1 で実行されたストア命令は 1 回であるため，T1 がアボートする場合はメモリへの書き戻しが 1 箇所で済む．したがって，このような場合は T2 よりも T1 をアボートした方がアボートコストは少ないと言える．

次に提案する LogTM がアボート対象を選択する動作について説明する．提案 LogTM において並列実行するトランザクションがアボート対象を選択する動作モデルを**図 5** に示す．図 5 では 2.2.2 項の図 3 の動作モデルと同様に，Thread1 は例 1 にあるトランザクション T1 を実行し，Thread2 は例 2 にあるトランザクション T2 を実行する．したがって図 5 t1 までのふたつのトランザクション内で実行される命令は図 3 t4 までに実行される命令と全く同じである．

アボートコストのより少ないトランザクションを選択するためには，デッドロック時にログエントリ数を比較する必要がある．そのため提案モデルではデッドロック時にアボート対象を選択する評価式を計算する．また，アボートの対象はログエントリ数のみに依存するため，トランザクション開始時刻を考慮する必要がなくなる．そのため，デッドロックを検知する条件を possible_cycle フラグがセットされているトランザクショ

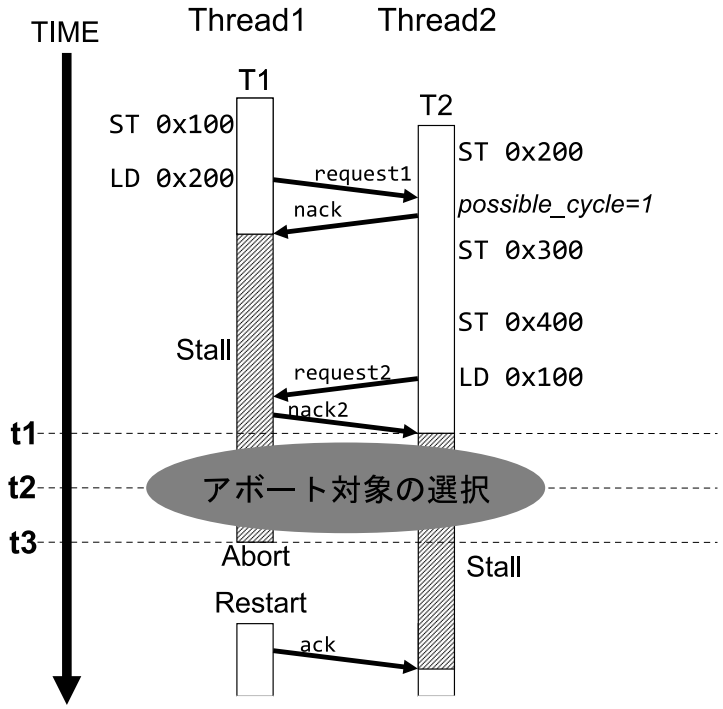

図 5: 既存の LogTM でのアボート対象の選択

ンが nack を受信したときとする．したがって図 5 では図 5 t1 で T2 が possible_cycle フラグをセットした状態で nack を受信するため，T2 は T1 とデッドロックに陥ったことを検知する．そしてデッドロックを検知した T2 は評価式を計算する (図 5 t2)．評価式はデッドロックしたトランザクションのログエントリ数を入力とする．また評価式はデッドロックしたトランザクションのどちらかで計算されるが，説明の簡略化のために省略する．評価式はログエントリ数の比較によって決定するため，以下ように定義できる．

$$Thrs = L1 - L2 \tag{1}$$

ここで評価式で計算される数式について説明する．ふたつのトランザクションがデッドロックに陥っており，どちらかをアボートさせなければならない場合を考える．評価式を計算してアボート対象を選択するトランザクションをトランザクション 1，トランザクション 1 とデッドロックに陥っているトランザクションをトランザクション 2 とする．トランザクション 1 が持つログエントリ数を $L1$ とし，同様にトランザクション 2 が持つログエントリ数を $L2$ とする．$L1 - L2$ はログエントリ数の差分を示す．$Thrs$

はどちらのトランザクションをアボートさせるかを示す．アボートの対象にはログエントリ数の小さい方が選択されるので，$Thrs$ が正のとき，$L1 > L2$ であるためトランザクション 2 がアボート対象として選択される．反対に $Thrs$ が負のとき，$L1 < L2$ であるためトランザクション 1 がアボート対象として選択される．

このような評価式が図 5 t2 で計算される．先ほど述べたように評価式はデッドロックしたトランザクションのどちらかでのみ計算されるが，評価式の結果によりアボート対象として選択されるトランザクションは不変である．例えば図 5 において T1 が評価式を計算する場合を考える．T1 が持つログエントリ数は 1 であり，T2 が持つログエントリ数は 3 であるため，評価式の入力パラメタは $L1 = 1, L2 = 3$ となる．したがって $Thrs < 0$ となるため，評価式を計算するトランザクション，すなわち T1 がアボートの対象として選択される．反対に T2 が評価式を計算する場合でも，評価式の結果は $Thrs > 0$ となり，T1 がアボートの対象として選択される．したがって評価式の結果により T1 はアボートする (図 5 t3)．このとき T1 のアボートコストは T2 に比べて小さいため，図 3 の例よりもアボートコストが削減できたといえる．

3.3 ログエントリ数とトランザクション開始時刻を用いたアボート対象の選択方法

前節で述べた手法はアボートコストのより少ないトランザクションをアボートさせるという単純なモデルであった．しかし 2.2.2 項で述べたように早く開始したトランザクションは早くコミットされることが望ましい．したがってアボートにおいてトランザクションの開始時刻は無視できないと考えられる．そこで本節では．アボートコストとトランザクション開始時刻を考慮したアボート対象を選択するハイブリッド型評価式を提案する．

トランザクション開始時刻を考慮するために，ハイブリッド型評価式内部でログエントリ数と開始時刻を比較する必要がある．そこで前項で提案した評価式の入力パラメタにデッドロックしたふたつのトランザクションの開始時刻を追加する．式 (1) と同様に，ハイブリッド型評価式を計算してアボート対象を選択するトランザクションをトランザクション 1，トランザクション 1 とデッドロックに陥っているトランザクションをトランザクション 2 と定義したとき，トランザクション 1 が持つ開始時刻のタイムスタンプを $T1$，トランザクション 2 が持つ開始時刻のタイムスタンプを $T2$ と定義する．また $L1 - L2$ はログエントリ数の差分を表し，$T1 - T2$ は開始時刻の差分を表す．ここでハイブリッド型評価式は以下のように定義できる．

$$Thrs = m(L1 - L2) - n(T1 - T2) \tag{2}$$

先ほど述べたように，ログエントリ数はより小さい方が，開始時刻はより大きい方がアボートされるべきである．これを一つの判定式で表現するため，式 (2) ではログエントリ数の差分に重み m を乗じたものから，開始時刻の差分に重み n を乗じたものを引いている．重み m 及び n が等しい場合，開始時刻の差に比べてログエントリ数の差の方が大きければログエントリ数によってアボート対象が選択される．反対に開始時刻の差の方が大きければ開始時刻によって選択される．ここで $Thrs$ が正になるのは，ログエントリ数によって選択される場合に $L1 > L2$ となるとき，あるいは開始時刻によって選択される場合に $T1 < T2$ となるときである．したがって $Thrs > 0$ のときにトランザクション 2 がアボートし，$Thrs < 0$ のときはトランザクション 1 がアボートする．

係数 m 及び n は，ログエントリ数とトランザクション開始時刻のどちらが重視するかを表す．例えば，$n = 0$ ならばトランザクション開始時刻は全く考慮されない．また，$m < n$ ならばトランザクション開始時刻を重視して選択される．

4 実装

本章では提案手法の実装について説明する．

4.1 ログエントリ数を用いたアボート対象の選択

ログエントリ数を用いたアボート対象を選択するために，3.2 節で述べたような評価式 (式 (1)) を計算する必要がある．評価式はデッドロックしたトランザクションのどちらか片方でのみ計算されため，評価式を計算するタイミングにより提案モデルの実装方法が大きく変わる．そこで以下の 2 種類タイミングで評価式を計算する提案モデルが考えられる．

nack を受信してデッドロックを検知した時:

nack を受信してデッドロックを検知したトランザクションが評価式を計算するモデルである．

nack を送信する前にデッドロックを判定した時:

nack を送信するトランザクションが評価式を計算するモデルである．nack を送信するトランザクションは，競合相手が nack を受信した場合にデッドロックを検

知するかどうかを nack を送信する前にあらかじめ知っておき，競合相手でデッドロックが検知されることを判定したら評価式を計算する．nack を送信したトランザクションは既にストールしており，ここで競合相手をストールさせるとデッドロックに陥る．したがって nack を送信するトランザクションは送信前にアボートさせる対象を選択する．

以降それぞれの実装方法について説明する．

4.1.1 nack 受信時に選択

本項では nack を送信するトランザクションが評価式を計算するモデルについて説明する．アボート対象を選択する評価式にログエントリ数を用いるために，計算中のトランザクションが保有するログの総エントリ数を把握する必要がある．そこで，キャッシュコントローラ上にログエントリ数のカウンタを設ける．このカウンタはスレッドごとに存在し，評価式を計算するトランザクションは自身の持つカウンタを参照することで自身のログエントリ数を知ることができる．拡張した提案手法のハードウェア構成を**図 6** に示す．図 6 では黒く塗りつぶされた部分が拡張した機構である．LogEntryCount はログエントリ数のカウンタを示し，Abort 及び Stall はそれぞれビットを表す．Abort 及び Stall の説明は後述する．

評価式を用いてログエントリ数を比較するためには，自身のログエントリ数だけでなくデッドロックした相手のログエントリ数を知る必要がある．そこで，プロセッサ・コア間の通信メッセージを拡張する．既存の LogTM では，プロセッサコア間の通信はメッセージキューで構成されており，プロセッサコア間の通信で送受信されるデータはすべてメッセージキューにバッファリングされる．したがって通信メッセージを拡張してログエントリ数を付加することで，メッセージ受信したトランザクションは相手のログエントリ数を知ることができる．

ここで nack 受信時に評価式を計算する場合の動作モデルを**図 7** に示す．図 7 の Thread1 では 3.2 節の例 1 にあるトランザクション T1 が計算され，Thread2 では 3.2 節の例 2 にあるトランザクション T2 が計算されるとする．したがって図 7 t2 までのふたつのトランザクション内で実行される命令はは図 3 t4 までに実行される命令と全く同じである．図 7 t3 では，T2 が T1 からの nack(図 7 nack2) を受信するためデッドロックを検知する．本手法では nack を受信してデッドロックを検知したトランザクションが判定関数を計算するため，T2 が評価式を計算する．先ほど述べたように，評価式を計算するためには，あらかじめデッドロックした相手のログエントリ数を知っておく必要がある．したがって T2 に対して T1 のログエントリ数を通知するために，T1 は

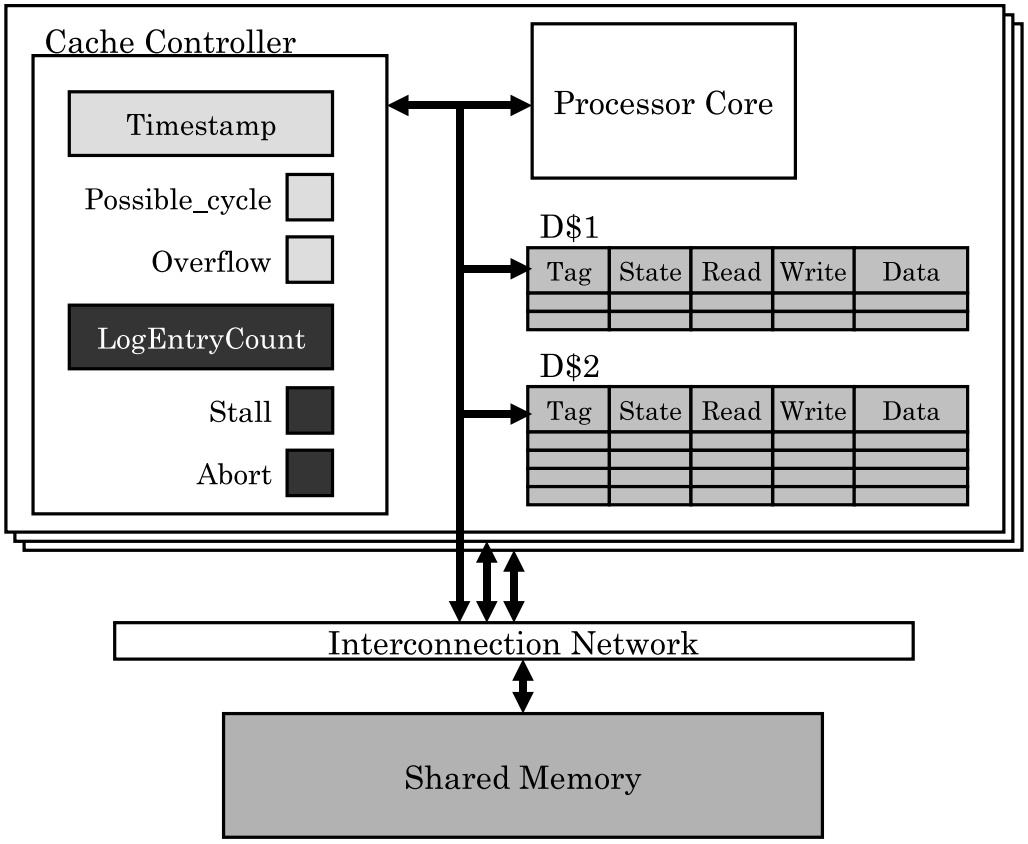

図 6: 拡張した LogTM の構成

送信する nack メッセージにログエントリ数を付加する (図 7 nack2)．これにより T2 は評価式を計算できる．

評価式では T1 と T2 のログエントリ数が比較され，アボートの対象が選択される．T1 で実行されたストア命令数は T2 で実行されたストア命令数に比べて少ないため，T1 のログエントリ数は T2 のログエントリ数と比べて小さい．したがって T1 がアボート対象として選択される．ここで T1 がアボートするためには，T1 がアボート対象として選択されたことを T2 から T1 に対して通知する必要がある．そこで通信メッセージを拡張し，新たに `abortreq` というメッセージを定義する．abortreq は T2 から T1 に対して送信され，abortreq を受信した T1 はアボートする (図 7 t5)．

しかし，既存の LogTM の仕様ではリクエストに対するレスポンスメッセージを受信する前にアボートすることはできないため，abortreq を受信したトランザクションはレスポンスメッセージを受信するまで待機する必要がある．そこで，トランザクションが abortreq を受信したことを管理するために，図 6 にあるようにキャッシュコントローラを拡張し Abort ビットを設ける．Abort ビットはトランザクションが abortreq

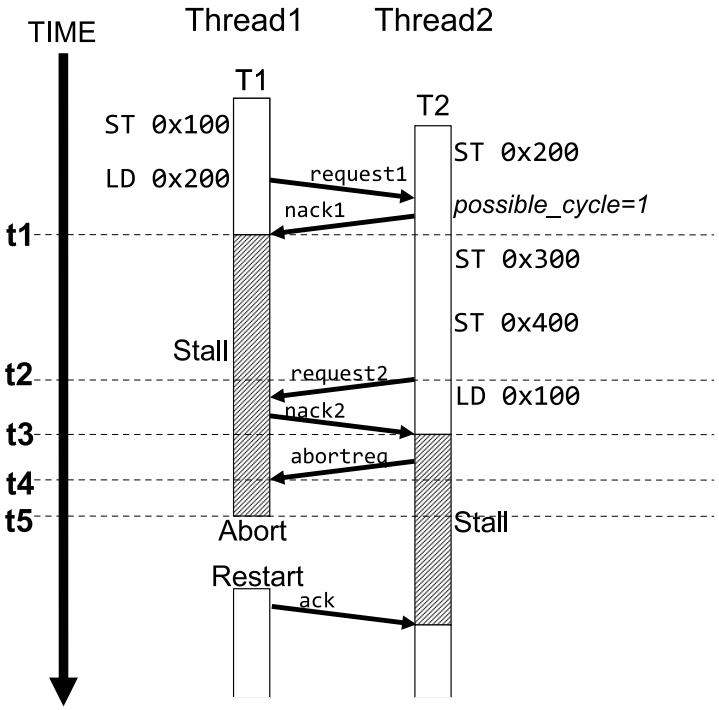

図 7: nack 受信時に評価式を計算

を受信したときにセットされる．図 7 では，T1 が abortreq を受信したときにセットされる (図 7 t4)．そして，Abort ビットがセットされたトランザクションが過去に送信したリクエストに対するレスポンスメッセージを受信した時にそのトランザクションはアボートする．したがって abortreq を受信してから実際にアボートするまでに時間差が生じる (図 7 t4–t5)．

また，ストールしていないトランザクションから nack を受信する場合，nack を受信したトランザクションの possible_cycle フラグがセットされているとデッドロックとして検出されて評価式が計算されてしまう．そのため，abortreq が送信されてストールしていないトランザクションがアボートしてしまう危険性がある．そこで nack 送信者は自身のストール状態を知らせるために，通信メッセージを拡張してストール状態をメッセージに付加する．さらにストール状態を管理するために，図 6 にあるようにキャッシュコントローラを拡張し Stall ビットを設ける．Stall ビットはトランザクションがストール状態になったときにセットされ，ストールから開放された時にクリアされる．図 7 では，T1 は nack を受信した時点でストール状態に移行するため，Stall ビットがセットされる (図 7 t1)．そして，ログエントリ数と同様に T1 は nack メッセージに T1 のストール状態を付加して T2 へ送信する (図 7 nack2)．これにより T2 は T1 が

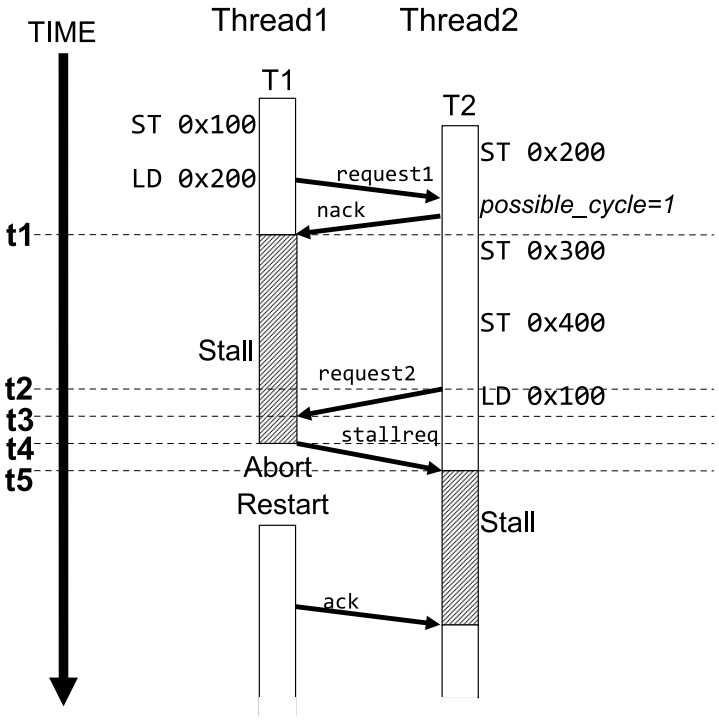

図 8: nack 送信時に評価式を計算

ストールしていることを知ることができる．

4.1.2 nack 送信時に選択

前項の手法では，abortreq を送信してアボートさせる場合，評価式によりアボート対象が判定されてから abortreq を受信するまで送信時間分の誤差が生じている．そこで，次にトランザクションが nack を送信する時にアボート対象を選択するモデルを考える．

前項の手法と同様に，評価式を用いてログエントリ数を比較する必要があるため，図 6 のようにハードウェアを拡張する．さらに，プロセッサ・コア間の通信メッセージを拡張してメッセージにログエントリ数を付加することで，トランザクションは互いのログサイズを知ることができる．

ここで nack 送信時にアボート対象を選択する場合の動作モデルを**図 8** に示す．図 7 と同様に，Thread1 では 3.2 節の例 1 にあるトランザクション T1 が実行され，Thread2 では 3.2 節の例 2 にあるトランザクション T2 が実行されるとする．さらに図 8 t1 までのふたつのトランザクションの動作は図 7 t1 までと同じように動作する．

図 5 の例と同様に，T2 は T1 からの nack(図 8 nack2) を受信するためデッドロックを検知する (図 8 t4)．ここで T1 が評価式を計算するためには，あらかじめ T2 のログエ

ントリ数を知っておく必要がある．さらに T1 は T2 に対して nack(図 8 nack2) を送信する前に T2 がデッドロックを検知することをあらかじめ知っておく必要がある．デッドロックが検知される条件は先ほど述べたように possible_cycle フラグがセットされているトランザクションがストールしたトランザクションから nack を受信したときである．したがって T2 は T1 に対してリクエストを送信すると同時に T2 のログエントリ数及び possible_cycle フラグの状態を送信することで，T1 は T2 がデッドロックしてしまう可能性を知ることができる (図 8 request2)．

また評価式を計算するためには，T1 がストールしている必要がある．そこで T1 のストール状態を管理するために，前項の手法と同様にキャッシュコントローラを拡張し Stall ビットを設ける．図 8 では，T1 は nack(図 8 nack) を受信した時点でストール状態に移行するため，Stall ビットがセットされる (図 8 t1)．これにより T1 はストール中にのみ評価式を計算することができる．

ここで評価式によって T1 がアボート対象として選択されたため，T1 はアボートする (図 8 t4)．しかし，前項で述べたように既存の LogTM の仕様ではリクエストに対するレスポンスメッセージを受信する前にアボートすることはできない．したがって前項の手法と同様に Abort ビットを設けることでトランザクションがレスポンスメッセージを待つ状態に移行する．図 8 の例では，T1 がリクエスト (図 8 request2) を受信したときに Abort ビットがセットされる (図 8 t3)．

一方で T2 は T1 のアボートが終了するまでストールする必要があるため，T1 は T2 がストールしなければならないことを通知する必要がある．そこで通信メッセージを拡張し，新たに `stallreq` というメッセージを定義する．stallreq は T1 から T2 に対して送信され，stallreq を受信した T2 はストールする (図 8 t5)．

4.2 ログエントリ数とトランザクション開始時刻を用いたアボート対象の選択

本節では，3.3 節で述べたログエントリ数とトランザクション開始時刻を用いたアボート対象選択方法について説明する．アボート対象の選択には，3.3 節のハイブリッド型評価式を用いる．ハイブリッド型評価式でトランザクションの開始時刻を比較するためには，デッドロックに陥ったトランザクションの開始時刻を知る必要がある．しかし，既存の通信メッセージには既にトランザクションの開始時刻が付加されているため，ハイブリッド型計算式を 4.1.1，4.1.2 項で述べたモデルが持つ評価式と置き換えるという簡単な方法で実装することができる．

表 1: システムモデルパラメータ

Processor	32 cores
周波数	1 GHz
	single-issue
	in-order
	non-memoryIPC=1
D1 cache	16 KBytes
ways	4 ways
latency	1 cycle
D2 cache	4 MBytes
ways	4 ways
latency	12 cycles
Memory	4 GBytes
latency	80 cycles
Interconnect Network	Hierarchical switching topology
link latency	14 cycles

5 評価

本章では，提案手法の有効性を示すために SPLASH-2 ベンチマーク [6] を用いて評価と考察を行う．

5.1 評価環境

評価にはフルシステムシミュレータである Virtutech Simics[7] と GEMS[8] を用いた．想定するシステムの環境を**表 1** に示す．シミュレーションにおいて Simics は機能シミュレーションを担当する．Simics ではシミュレーションのターゲットモデルとして SPARC V9 ISA を選択した．また，OS は Solaris10 を用い，各プロセッサは単命令発行のインオーダ実行である．GEMS はメモリシステムの詳細なタイミングのシミュレーションを行う．

また評価対象のプログラムには共有メモリ型並列計算用のベンチマークプログラムである SPLASH-2 ベンチマークプログラムの内 Barnes と Raytrace を用いた．それぞれの入力を**表 2** に示す．

表 2: SPLASH-2 ベンチマークプログラムとその入力

Barnes	512 bodies
Raytrace	small image (teapot)

表 3: Barnes 内で計測された差分

ログエントリ数の差分	最大値	39.00
	最小値	3.00
	平均	7.96
トランザクション開始時刻の差分	最大値	961014.00
	最小値	1.00
	平均	7863.14

5.2 評価結果

評価結果のグラフは，左から順に

LogTM ：既存の LogTM

Hybrid1(m,n) ：4.1.1 項の提案手法をベースにしたハイブリッド型

Hybrid2(m,n) ：4.1.2 項の提案手法をベースにしたハイブリッド型

での実行結果を示している．ハイブリッド型では係数 (m, n) を複数設定する．

まず SPLASH-2 ベンチマークの内，Barnes の評価結果を**図 9** に示す．図 9 は Barnes での実行サイクル数を示し，それぞれ LogTM を 1 として正規化した．ハイブリッド型モデルの係数として，まず $(m : n) = (1 : 0), (0 : 1)$ を選択した．$(m : n) = (1 : 0)$ はログエントリ数のみを考慮し，$(m : n) = (0 : 1)$ はトランザクションの開始時刻のみを考慮するモデルに相当する．ここで，既存モデルである LogTM 上で Barnes を実行したときに，デッドロックに陥ったふたつのトランザクションが持つログエントリ数とトランザクション開始時刻を全て観測し，それぞれの差分を計測した結果を**表 3** に示す．この結果から，ログサイズ差と開始時刻差の平均値を比較すると $O(1000)$ の差が生じていることがわかる．したがってログエントリ数と開始時刻を比較する条件を一致させるために，ハイブリッド型モデルの係数セットとして $(m : n) = (1000 : 1)$ も採用した．

図 9 の結果より，既存手法と比べて全ての提案手法で実行サイクル数の削減が認められる．特にログエントリ数を重視した手法である Hybrid1$(1, 0)$，Hybrid2$(1, 0)$ でより多く削減できたことがわかる．また，Hybrid1$(0, 1)$ と Hybrid2$(0, 1)$ は LogTM とほ

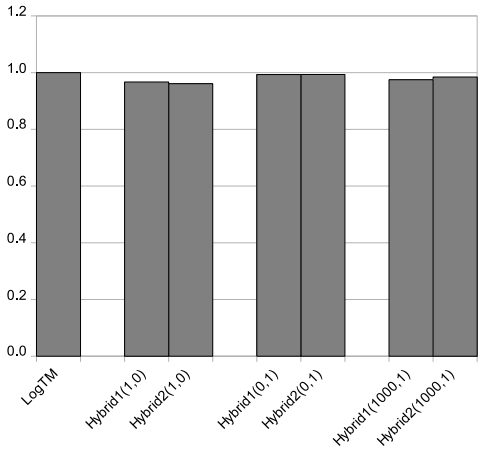

図 9: Barnes での
実行サイクル数の評価結果

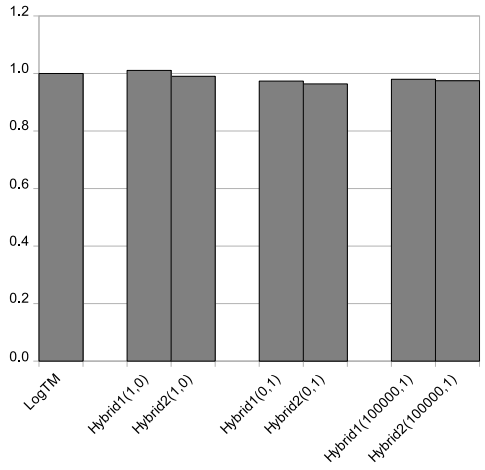

図 10: Raytrace での
実行サイクル数の評価結果

ぼ同じサイクル数となっている．これは LogTM と同様にトランザクション開始時刻のみを考慮しているからである．しかし，提案手法では 3.2 節で述べたように LogTM とは異なるタイミングでアボート対象を選択しているため，サイクル数に若干の差が生じている．

ここでアボート時にログから書き戻されたエントリ数を計測した結果を**図 11** に示す．それぞれ LogTM で書き戻されたログエントリ数を 1 として正規化している．図 11 の結果と照らし合わせると，実行サイクル数が削減したモデルの全てで書き戻されたログエントリの総数が減少しており，このことからアボートサイクル数の減少によって実行サイクル数が削減したことがわかる．

次に Raytrace の実行サイクル数を**図 10** に示す．やはり，それぞれ LogTM の実行サイクル数を 1 として正規化している．ハイブリッド型モデルの係数は Barnes と同様に $(m:n) = (1:0), (0:1)$ を選択した．ここで表 3 と同様にデッドロックに陥ったふたつのトランザクションが持つログサイズと開始時刻をすべて観測し，それぞれの差分を計測した結果を**表 4** に示す．その結果，それぞれの平均を比較すると $O(100000)$ の差が生じていることがわかる．したがってハイブリッド型モデルの係数セットに $(m:n) = (100000:1)$ を採用する．

既存手法と比べて，ログエントリ数のみを考慮した手法 Hybrid1$(1,0)$，Hybrid2$(1,0)$ では実行サイクル数が LogTM とほぼ同じか増加してしまっていることがわかる．ここでアボート時にログから書き戻されたエントリ数を計測した結果を**図 12** に示す．そ

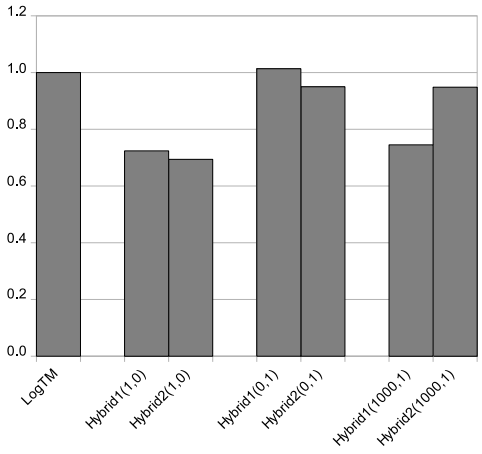

図 11: Barnes での書き戻された
ログエントリ数の評価結果

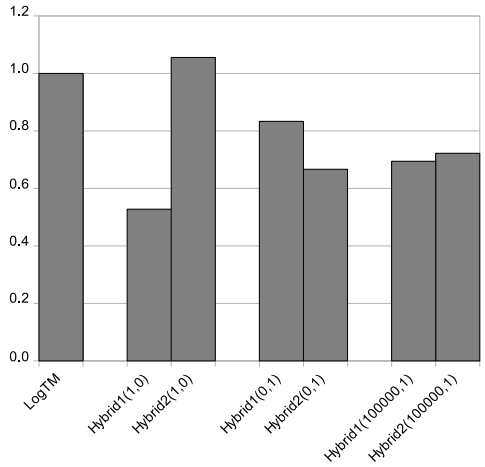

図 12: Raytrace での書き戻された
ログエントリ数の評価結果

表 4: Raytrace 内で計測された差分

ログエントリ数の差分	最大値	1.0000
	最小値	0.0000
	平均	0.0098
トランザクション開始時刻の差分	最大値	961014.0000
	最小値	1.0000
	平均	5967.2400

れぞれ LogTM で書き戻されたログエントリ数を 1 として正規化している．図 12 の結果と照らし合わせると，Hybrid1$(1,0)$ で書き戻されたログエントリの総数が減少していることがわかる．このことから，実行サイクル数が削減できない場合でも，アボートサイクル数を削減できていることがわかる．

一方でトランザクションの開始時刻のみを考慮する手法 Hybrid1$(0,1)$，Hybrid2$(0,1)$ では実行サイクル数の削減が認められた．したがって Raytrace ではログエントリ数の小さい方をアボートしても効果が得られないことがわかる．しかしながら，ログエントリ数と開始時刻の両方を考慮する手法 Hybrid1$(100000,1)$，Hybrid2$(100000,1)$ では実行サイクル数を削減することができた．この理由については後述する

結果をまとめると，SPLASH-2 ベンチマークの Barnes と Raytracen の両方で既存の LogTM に比べて最大で約 4%，平均で約 2%の実行サイクル数が削減できた．

5.3 考察

Barnes は競合したトランザクションの保有するログエントリ数の差の絶対値が大きいプログラムであるため，トランザクションのアボートコストにも大きな差が生じる．したがってアボートにかかるサイクル数が減少したため，全体の実行サイクル数が削減されたと考えられる．

一方で，Raytrace はログエントリ数の小さい方をアボートさせる手法を用いても実行サイクル数を削減することができなかった．これは Raytrace のようなログエントリ数の差の絶対値が非常に小さいプログラムはそもそもアボートコスト自体が小さいため，アボートコストの小さい方をアボートさせる手法を用いてもあまり効果が表れない．さらに，表 4 からもわかるようにログエントリ数の差の絶対値に比べてトランザクションの開始時刻の差の絶対値が非常に大きいため，わずかなログサイズの違いにより非常に長い期間実行しているトランザクションがアボートされてしまう可能性がある．つまり長い期間実行する必要があるトランザクションがコミットされずに残り続けることで，かえって競合が頻発してしまうことが考えられる．このことから，ログエントリ数の小さいトランザクションを選択してアボートサイクル数を削減できたとしても，競合によって長時間ストールすることで実行サイクル数が増加してしまうと考えられる．以上より，Raytrace のようなプログラムはログエントリ数の小さいものをアボートさせる手法を実行するモデルにとって最も不向きなものであると言える．

このように，ログエントリ数のより小さいトランザクションをアボートさせる手法は，競合したトランザクションの保有するログエントリ数の差が大きいプログラムを実行した場合に有効に働く．反対にログサイズの差が小さいプログラムでは大きな効果は期待できない．

しかし，ログエントリ数とトランザクションの開始時刻の両方を考慮したハイブリッド型の手法を用いることでログサイズ差の小さいプログラムでも実行サイクル数を削減することができる．これは，ログサイズの差がわずかな場合にログサイズの小さい方ではなく開始時間の遅い方をアボートさせることで，余分な競合の発生を抑えることができるためである．したがってハイブリッド型評価式にログサイズと開始時刻を同じ条件で比較できるような最適な係数をを設定することで，より一般的なプログラムを用いても実行サイクル数を削減することができる．

しかしながら，そのような最適な係数はプログラムごとに異なっているため，プログラマが最適な係数を設定することは難しい．そこで今後の課題として，競合するふたつのトランザクションのログサイズ差と開始時刻差をプログラムの実行中に観測し，

動的に最適な係数を設定する手法が考えられる．

6　おわりに

本研究では，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，アボート時のメモリへの書き戻しコストを削減する手法を提案した．拡張した LogTM ではアボート対象の選択にログエントリ数を用い，ログエントリ数の小さい，すなわちアボートコストの少ないトランザクションを動的にアボート対象として選択することで，プログラム全体の実行時間を削減した．また，アボート対象の選択にトランザクションの開始時刻も考慮したハイブリッドな手法を提案した．提案手法の有効性を確認するため，SPLASH-2 ベンチマークのうち，Barnes と Raytrace を用いた評価を行った．その結果，Barnes と Raytrace の両方で既存の LogTM に比べて最大で約 4%，平均で約 2%の実行サイクル数が削減できた．

今後の課題として，ハイブリッド手法で用いたアボート対象を選択する評価関数の係数の値を動的に決定することが挙げられる．競合したトランザクション同士のログエントリ数の差とトランザクションの開始時刻の差はプログラムごとに異なるため，ログエントリ数と開始時刻の条件を同一にするような最適な係数を動的に設定する必要がある．この手法により一般的なプログラムに対応することができると考えられる．

謝辞

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名古屋工業大学の松尾啓志教授，齋藤彰一准教授，松井俊浩助教，そして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，終止熱心に御指導頂いた津邑公暁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・津邑研究室及び齋藤研究室の先輩，同期，そして研究グループ内の方々に深く感謝致します．ありがとうございました．

参考文献

[1] Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D. and Wood, D. A.: LogTM: Log-based Transactional Memory, *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, IEEE Computer Society, pp. 254–265 (2006).

[2] Herlihy, M. and Moss, J. E. B.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ACM, pp. 289–300 (1993).

[3] Sweazey, P. and Smith, A. J.: A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus, *ISCA '86: /raytraceProceedings of the 13th Annual International Symposium on Computer Architecture*, IEEE Computer Society Press, pp. 414–423 (1986).

[4] Censier, L. M. and Feautrier, P.: A New Solution to Coherence Problems in Multicache Systems, *IEEE Transactions on Computers*, IEEE Computer Society, pp. 1112–1118 (1978).

[5] Rajwar, R. and Goodman, J. R.: Transactional Lock-Free Execution of Lock-Based Programs, *Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 5–17 (2002).

[6] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P. and Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations, *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, ACM, pp. 24–36 (1995).

[7] Magnusson, P. S. and et al: Simics: A Full System Simulation Platform, *Computer*, Vol. 35, pp. 50–58 (2002).

[8] Martin, M. M., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., E.Moore, K., Hill, M. D. and Wood., D. A.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset., *Computer Architecture News*, ACM, pp. 254–265 (2005).