卒業研究論文

柔軟なチェックポインティングによる LogTMの効率的ロールバック

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 19 年度入学　19115021 番

江藤 正通

平成 23 年 2 月 8 日

柔軟なチェックポインティングによるLogTMの効率的ロールバック

江藤 正通

内容梗概

マルチコア環境におけるスレッドレベル並列性 (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る場合では，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 LogTM が提案されている．

LogTM はトランザクショナル・メモリの一種であり，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破棄する．この操作をアボートという．トランザクションの再実行を行うために，トランザクション開始時にそのときの状態を保持しておく．LogTM では，そのトランザクション開始位置をチェックポイントとし，アボートしたトランザクションを競合したアドレスよりも前に通過したチェックポイントの状態に戻し，再実行する．また，プログラマがトランザクション内部に別のトランザクションを指定することで，トランザクション内部にチェックポイントを作成することもできる．競合したメモリアクセスの時刻がトランザクション中のチェックポイント作成時刻より後なら，トランザクションの途中から再実行する．しかし，LogTM ではトランザクション開始位置からのみ再実行可能であるため，プログラマがどのようにトランザクションを定義するかによって再実行の開始位置が変化し，これが性能に影響を与える場合がある．例えば，長いトランザクションを実行している場合にアボートすると，再実行する命令が多くなり性能の低下につながってしまう．本論文ではこのような問題を解決するために，トランザクションの途中に動的にチェックポイントを作成する手法を提案する．これにより，トランザクションの途中から再実行する可能性が増え，再実行コストを減らすことができる．さらに，チェックポイントの作成条件をトランザクションの実行中に動的に変更する手法も提案する．これにより，様々な長さのトランザクションに対応することができる．

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，シミュレーションによる評価を行った．評価の対象として付属ベンチマークプログラムの contention，prioque，slist を選択した．その結果，再実行コストは削減できたが，ログの書き戻しコスト等が増加する結果となった．

柔軟なチェックポインティングによるLogTMの効率的ロールバック

目次

1 はじめに **1**

2 LogTM **2**

2.1 トランザクショナル・メモリの基本概念 2

2.2 データのバージョン管理 3

2.3 競合検出 5

2.4 部分ロールバック 11

3 提案 **14**

3.1 LogTMの問題点 14

3.2 ロード/ストア回数に基づくチェックポイントの作成 18

3.3 チェックポイント間隔の決定手法 18

4 実装 **20**

4.1 ロード/ストアカウンタとチェックポイントの操作 20

4.1.1 拡張したLogTMの構成 20

4.1.2 ロード/ストアカウンタ 21

4.2 動的チェックポイントの削除条件 21

4.3 チェックポイント間隔の動的変更 22

5 評価 **23**

5.1 評価環境 23

5.2 評価結果 24

5.3 考察 26

6 おわりに **27**

謝辞 **28**

参考文献 **28**

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 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．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．しかしながら，消費電力の増加や配線遅延の相対的な増加という問題から，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．例えば，スレッドレベル並列性 (Thread-Level Parallelism: TLP) を利用して並列に実行することで，プログラム全体の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並列プログラミングが一般的であるが，共有リソースに対して同期をとる必要があり，排他制御機構として一般的にロックが用いられている．しかし，ロックを用いた場合では，ロック操作に要する処理にオーバヘッドが発生し，また並列性の低下やデッドロックの発生などの問題も生じる．ロックによるオーバヘッドを削減するためには，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，ロックを用いない並行性制御機構として LogTM[1] が提案されている．

LogTM は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，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．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破棄する．この操作を**アボート**という．トランザクションを再実行するために，トランザクション開始時にそのときの状態を保持しておく．これを**チェックポイント**と言う．アボートしたトランザクションは，競合したアドレスへのアクセス時刻よりも前に作成したチェックポイントまで戻って再実行する．このようにして LogTM は共有リソースの排他制御を実現している．また，トランザクションの実行途中にチェック

ポイントを作成することもできる．もし，競合したアドレスへのメモリアクセス時刻がトランザクション内部のチェックポイント作成時刻よりも後なら，そのチェックポイントの位置から再実行する．しかし，LogTM はアボート時の再開位置をプログラマが指定したトランザクションの開始位置としているので，再実行コストが高くなってしまう可能性がある．例えば，長いトランザクションを実行している場合にトランザクションの後半部分でアボートすると，再実行する命令が多くなり性能の低下につながってしまう．本論文ではこのような問題を解決するために，トランザクションの途中に動的にチェックポイントを作成する手法を提案する．これにより，トランザクションの途中から再実行する可能性が生まれ，再実行コストを減らすことが期待できる．さらに，トランザクションの性質に合わせるためにチェックポイントの間隔を変更する手法も提案する．これにより，トランザクションの長さに合わせてチェックポイントの間隔が変更され，チェックポイント数が増大しすぎるのを防ぐことができる．また再実行コスト削減も期待できる．

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 章では本提案手法の評価を行い，最後に，6 章で結論を述べる．

2　LogTM

本章では，本研究の対象となるトランザクショナル・メモリの基本的概念とその一種である LogTM について説明する．

2.1　トランザクショナル・メモリの基本概念

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，共有メモリ型の手法が多く用いられる．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ため，同一のアドレスに対してアクセスするには調停をとる必要がある．そのような排他制御機構として一般的にロックが用いられている．しかし，ロックを用いた調停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が増加すると，ロックの獲得・解放に伴うオーバヘッドも増加し，性能が低下する可能性もある．さらに，プログラムごとに最適な粒度を設定するのは難しく，これら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．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要件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，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監視する．そして，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するメモリアドレスが同一である場合，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開始時点から再実行される．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向上するため，コア数に応じて性能が向上しやすくなる．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，プログラミングが比較的容易になる．

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，及びコミットをハードウェア的にサポートする．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．これにより，アボート時にはキャッシュ上の値を破棄するだけでトランザクションを開始時点の状態から再実行することができる．

2.2 データのバージョン管理

LogTMでは，トランザクションは投機的に実行される．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ことに割り当てる．

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が，その際，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しておく．これによりアクセスしたデータの古いバージョンを管理することができる．

なお，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，トランザクションのアボートにはトランザクション開始時点でのメモリ状態がわかれば十分だからである．

LogTMでのバージョン管理の様子を**図1**に示す．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する様子を(a)から(d)に示している．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．図1(a)はトランザクションの開始時点の様子を示している．このとき，メモリの0x100番地には既に値10が格納されている．トランザクションが進行し，図1(b)のように`ST 0x100,15`が実行されると，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．

次に投機的実行が成功した場合を図1(c)に示す．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．

一方で投機的実行が失敗した場合を図1(d)に示す．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破棄する．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

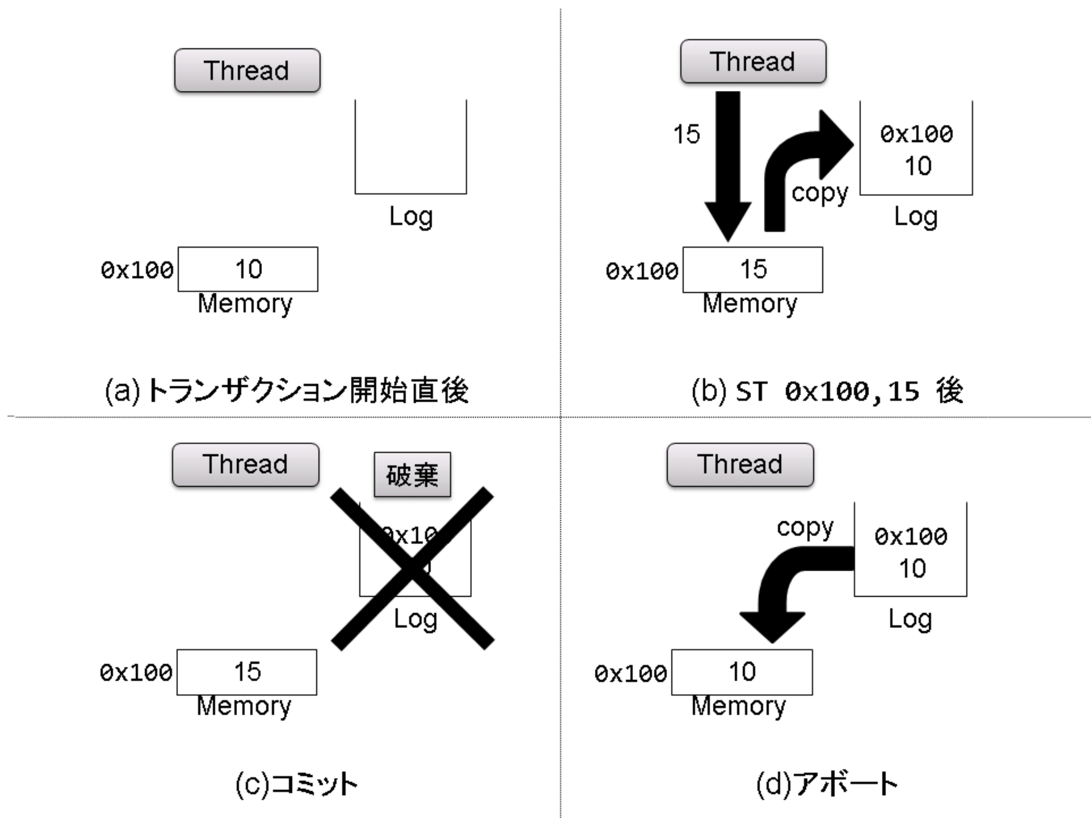

図 1: データのバージョン管理

する．つまり LogTM は，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．

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ことにより再実行を可能にしている．その方法として，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の状態等を保存し，その状態を**チェックポイント**として登録しておく．そして，アボート時にチェックポイントを用いることにより，レジスタの状態等をトランザクションの開始時と同じ状態に戻している．

2.3 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．したがって，メモリアクセスの競合検出を行うためにどのメモリアドレスがトランザクション内でアクセスされたかを記憶しておく必要がある．そこで，**図 2** のように，LogTM ではキャッシュライン上に新たに read セット (R) 及び write セット

Address	R	W	Value
0x100	1		10
0x200		1	20

図 2: リードライトセット

(W) 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 0x100 番地にリードアクセス，0x200 番地にライトアクセスが発生しているとすると，アクセスのあったラインに対して R に read ビットが格納され，W に write ビットが格納される．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リセットされる．

また，LogTM ではトランザクションに競合が発生したことを通知するため，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 [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，メモリアクセスを行うスレッドが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 セット及び write セットを参照する．これにより，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トランザクションによって既にアクセスされているか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ため，トランザクションの要件を満たさない．

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ため，トランザクションの要件を満たさない．

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書き込み対象アドレスの値が既に変更されているため，トランザクションの要件を満たさない．

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．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送信されるが，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．このとき，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状態を知ることができる．一方で競合が発生しなかった場合は ACK が返信される．

トランザクションが並列に実行して競合を検出する動作モデルを**図 3** に示す．図 3 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 3 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，図 3(a) を用いて競合が発生しない場合について説明する．図 3(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 番地に対するリクエストを送信する．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．Thread2 は命令を実行する前に 0x100

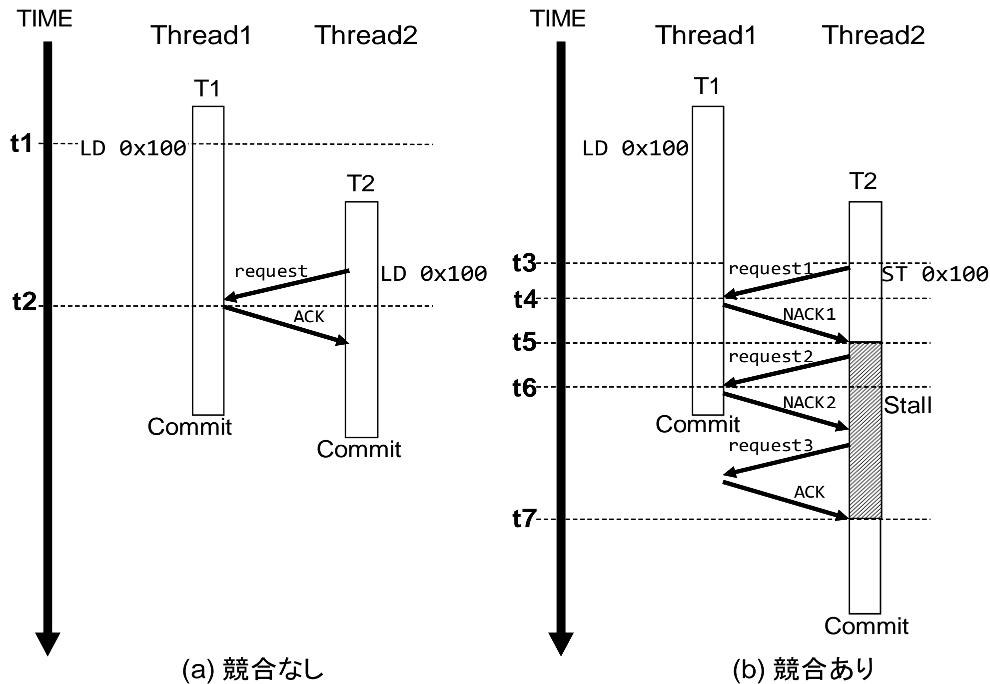

図 3: 競合の検出

番地に対するリクエストを送信する．図 3 中の `request` は送信されたリクエストを示している．時刻 t2 で Thread1 はリクエストによって Thread2 が 0x100 番地にアクセスしようと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，Thread1 は Thread2 に対して ACK を返信する．そして，ACK を受信した Thread2 は命令を実行することができる．

次に，図 3(b) を用いて競合が発生する場合について説明する．図 3(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．よって Thread2 からのリクエストを受信した Thread1 は時刻 t4 で競合したことを通知するために Thread2 へ NACK を返信する．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 request3 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．した

がって，時刻 t7 で Thread2 はトランザクションをストール状態から復帰させ，トランザクションの実行を続けることができる．

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場合がある．例えばトランザクション T1 と T2 が投機的に実行されていると仮定する．今，T1 から T2 に NACK を返信し，その後 T2 から T1 に NACK を返信したとする．このときお互いがお互いの終了を待ってしまい，一種のデッドロック状態になる．このようなデッドロック状態を解消するために，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．これは，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と考えられ，このトランザクションを早くコミットすることで競合の頻発を防ぐことができるからである．

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には，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持している．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 フラグを保持する．possible_cycle フラグは TLR's distributed timestamp method[5] に基づいた手法である．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 を返信したとき，送信側のトランザクションに 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 4** に示す． ここで，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，Thread1 が実行を開始した後に Thread2 が実行を開始する．次に Thread1 が `ST 0x100` を実行し，その後に Thread2 が `ST 0x200` 実行したとする．このときのリクエストは図中では省略している．さらにその後，時刻 t1 で Thread1 が 0x200 番地に対するロード命令を実行しようとする．このとき，Thread1 は命令を実行する前に Thread2 に 0x200 番地に対するリクエストを送信する．リクエストを受信した Thread2 は競合したことを通知するため Thread1 へ NACK を返信する．時刻 t2 において，Thread2 は自分よりも開始時刻の早いトランザクションを実行している Thread1 へ NACK を返信したので，Thread2 の possible_cycle フラグ

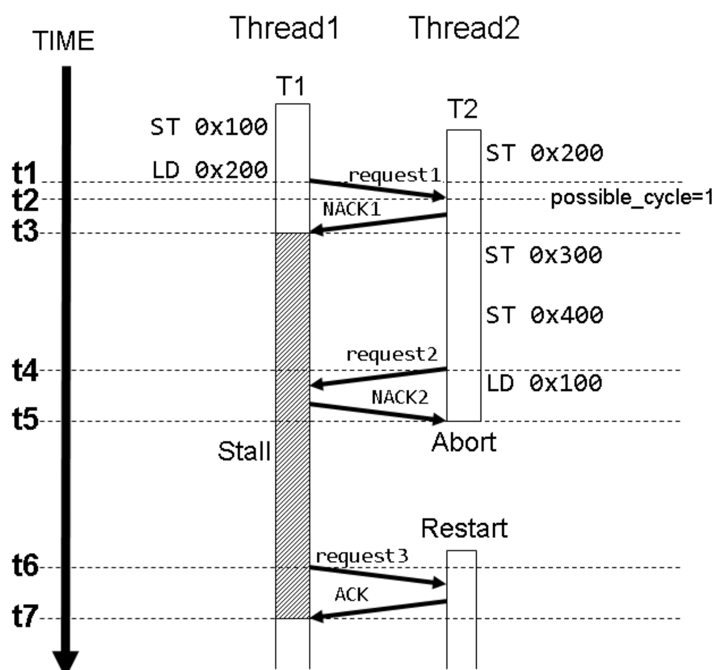

図 4: 既存の LogTM でのアボート対象の選択

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
```

がセットされる．そして，Thread1 は時刻 t3 で NACK を受信し，トランザクションをストールさせる．その後，Thread2 で `ST 0x300`，`ST 0x400` が実行されるが，Thread1 がアクセスしたアドレスと競合しないため処理が進む．さらにその後，Thread2 が時刻 t4 で 0x100 番地に対するロードを実行しようとする．しかし Thread1 と競合するため，Thread2 は時刻 t5 で NACK を受信する．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 Thread1 から NACK を受信し，かつ Thread2 には pos-

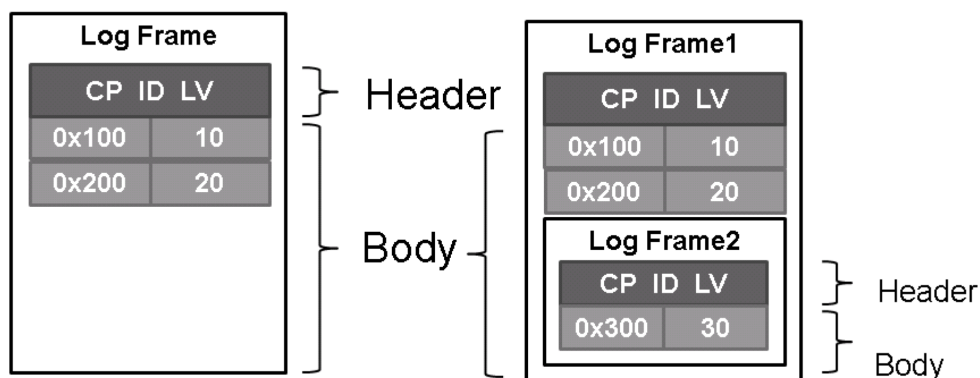

図 5: ログフレームの構成

sible_cycle フラグがセットされているため，Thread2 は T2 をアボートする．そして，Thread2 は時刻 t6 で開始時点の状態から再実行する．また，T2 がアボートしたことにより Thread1 で 0x200 番地にアクセスできるようになるため，時刻 t7 において T1 のストール状態が解消される．

2.4　部分ロールバック

LogTM では，アボートすることによって実行結果を破棄した状態から，再実行可能な状態に戻るためにログに退避した値をキャッシュに書き戻す必要がある．ログは，チェックポイントやトランザクションを識別するための ID 等を記録するヘッダ部，ライトアクセスによって書き換わる前の値とそのアドレスを記録するボディ部から成る．また，ログの領域全体を**ログフレーム**と言う．ログフレームは**図 5**(a) のように構成されている．トランザクションを開始すると，ログのヘッダ部が作成される．そして，書き戻しに必要な値とそのアドレスがボディ部の図中上から下方向に順に積まれていくスタック構造となっている．以下，ログの保存とログからの書き戻しについて説明する．

ログの保存:

トランザクションを開始すると，図 5 の Header のようにログフレームのヘッダ部にチェックポイント (CP) やトランザクション ID(ID)，及びトランザクションレベル (LV) が保存される．トランザクション ID とはプログラマが各トランザクションに与える固有の ID である．トランザクションレベルとは，トランザクションのネストの深さを表している．すなわち，トランザクションが開始された時点で

レベル 1 となり，そのトランザクション実行中に新たなトランザクションが開始されチェックポイントが作成されると，レベル 2 となる．このレベルは新たなトランザクションが開始されるたびにインクリメントされ，トランザクションがコミットされるたびにデクリメントされる．トランザクションがアボートされた際は，当該トランザクションの開始時点でのトランザクションレベルに戻る．ログフレーム内に記憶されるのは，チェックポイントが作成された時点でのトランザクションレベルである．

また，あるキャッシュラインに対してトランザクション内で初めてストア命令が実行されると，図 5(a) の Body のように，変更前の値とそのアドレスが保存される．この図では，トランザクション開始後に 0x100 番地に書き込みが行われ，その書き込み前の値は 10 であったこと，またその後，0x200 番地に書き込みが発生し，その書き込み前の値が 20 であったことがわかる．

ログからの書き戻し:

アボート処理では実行結果を破棄し，ログを用いて状態を復元する．ログからの書き戻しはログの書き込みとは逆順に行う．図 5(a) を用いて説明すると，まず 0x200 番地の値 20 をキャッシュに書き戻し，次に 0x100 番地の値 10 をキャッシュに書き戻す．そして，ログのヘッダ部に到達すると，書き戻し処理をこれ以上する必要がないとわかる．最後にレジスタ等をトランザクション開始時と同じ状態に戻す．この一連の書き戻し処理を**ロールバック**という．ロールバックしたスレッドは当該トランザクションを再実行する．

以上のようにして，ログを用いてトランザクション開始時の状態を復元している．しかし，このように毎回トランザクションの開始点までロールバックするのでは再び実行する命令が多くなってしまう．そこで，LogTM では**部分ロールバック**をサポートしている．トランザクションの実行中に新たなトランザクションに到達すると，部分ロールバックを行うために図 5(b) のようにログにその時の状態を記録する．このように，トランザクション内部でチェックポイントを作成すると，当該チェックポイントへのロールバックに用いるログフレームが作成され図 5(b) の Log Frame2 のようにログフレームが入れ子となる．このようにトランザクションの内部にトランザクションがあるとき，どのチェックポイントへロールバックすれば良いか判定する必要がある．そこで，図 2 で説明したリードライトセットを拡張してどのチェックポイントまで状態を復元するか判定している．

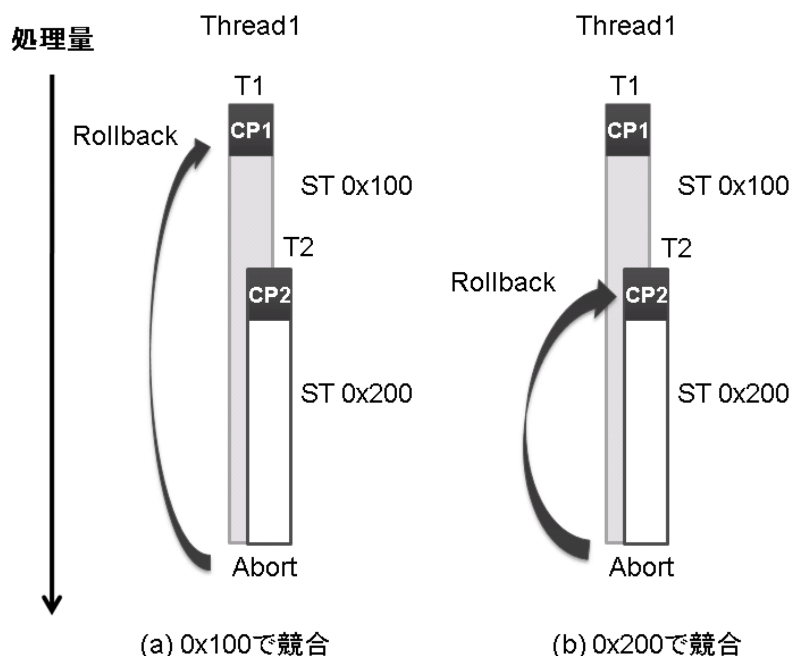

図 6: 再実行開始位置の選択

図 6 を用いて再実行開始位置の選択方法について説明する．ここで CP はチェックポイントを示している．LogTM での再実行開始位置は図 6 のように，競合したアドレスへのアクセス以前のチェックポイントとしている．これによりトランザクションの性質を満たし，キャッシュの一貫性を保つことができる．例えば図 6(a) のように，0x100 番地に対する競合の場合では CP2 以前のアクセスなので CP1 へロールバックし，図 6(b) のように，0x200 番地に対する競合の場合では CP2 以後のアクセスなので CP2 へロールバックすれば良い．

次に，再実行開始位置のチェックポイントを特定する方法を説明する．**図 7** のようにそれぞれのキャッシュラインに対して，まだ削除されていないチェックポイントの数だけ read セットと write セットを用意する．この R や W の後ろにある数字はトランザクションレベルに対応するため，どのトランザクションレベル内でアクセスしたのか判定することができる．**図 8** のように時刻 t1 でトランザクションが開始され，時刻 t2 で 0x100 番地にストア命令を実行したとする．このとき，トランザクションレベルは 1 なのでリードライトセットの 0x100 番地の W1 にビットがセットされる．次に時刻 t3 で新たなトランザクションを実行するとトランザクションレベルは 2 となる．時刻 t4 で 0x200 番地にロード命令が発生すると，0x200 番地のトランザクションレベル 2 に対応する R2 にビットがセットされる．時刻 t5 の 0x200 番地にストア命令が発生

Address	R1	W1	R2	W2	Value
0x100		1			10
0x200			1	1	20

図 7: 拡張したリードライトセットの利用

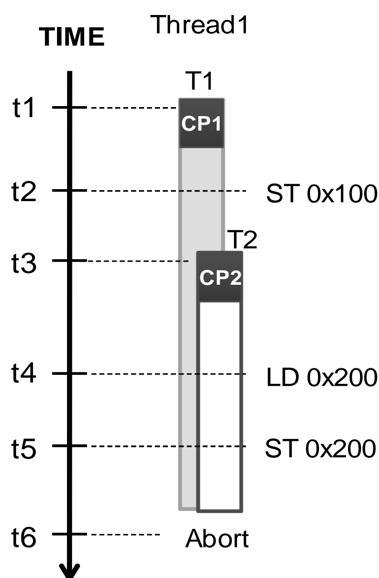

図 8: トランザクションの実行例

した場合も同じトランザクションレベル 2 なので W2 にビットがセットされる．ここで 0x200 番地に対する競合が発生し，トランザクションをアボートするとする．リードライトセットからは，トランザクションレベル 1 の中では 0x200 番地へアクセスしておらず，トランザクションレベル 2 の中でアクセスで競合しているとわかるので，2 つ目のチェックポイントまでロールバックすれば良いと判定できる．このようにして再実行開始位置のチェックポイントを特定している．

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．

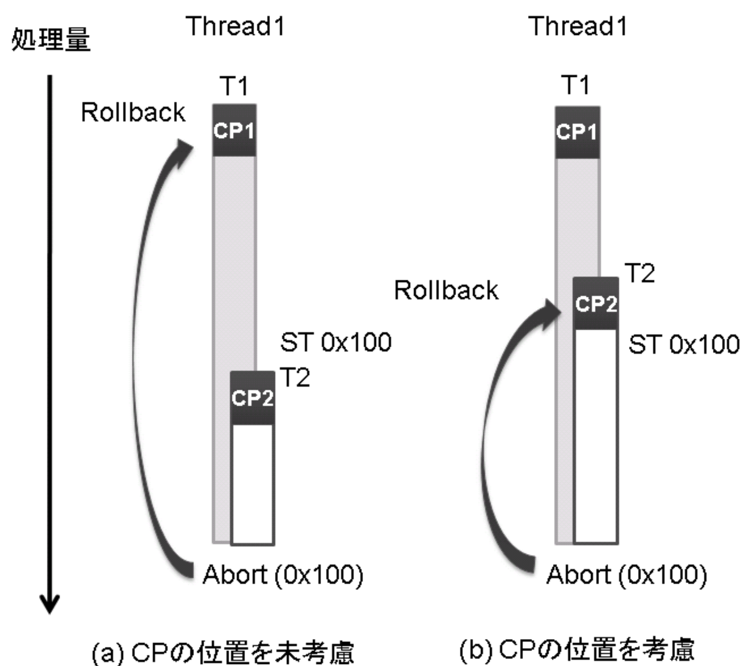

図 9: チェックポイントの位置の考慮

3.1 LogTM の問題点

既存の LogTM では，プログラマがトランザクションの範囲を指定する必要がある．そして，チェックポイントが作成されるのはそのトランザクション開始位置である．しかし，トランザクション内の適切な位置に新たなトランザクションを設定するには，トランザクション内のどの命令が競合しやすいのかという，トランザクションの性質を把握しなければならない．**図 9** に 0x100 番地において競合が発生しやすい場合の例を示す．プログラマが競合の発生しやすい位置を考慮せずに図 9(a) のようにトランザクションを設定すると，CP2 に戻ることのできる場面は少なく，CP2 のためのコストが無駄になる．一方，プログラマがトランザクション内のどの命令が競合しやすいかを把握して図 9(b) のようにトランザクションを設定すると，0x100 番地の直前から再実行できる可能性が高くなる．しかし，トランザクションの性質を考慮してトランザクションを設定するのは難しい．

このように，プログラマによるトランザクションの範囲指定が適当でないとチェックポイントの間隔が長くなる場合と短くなる場合がある．まず，長くなる場合の問題について説明する．例えば，**図 10**(a) のようにトランザクション T2，T3 を設定していれば，途中から再実行できる場合でも，図 10(b) のように長いトランザクションの途中にトランザクションを設定しない場合では，アボート時に多くの実行内容が破棄

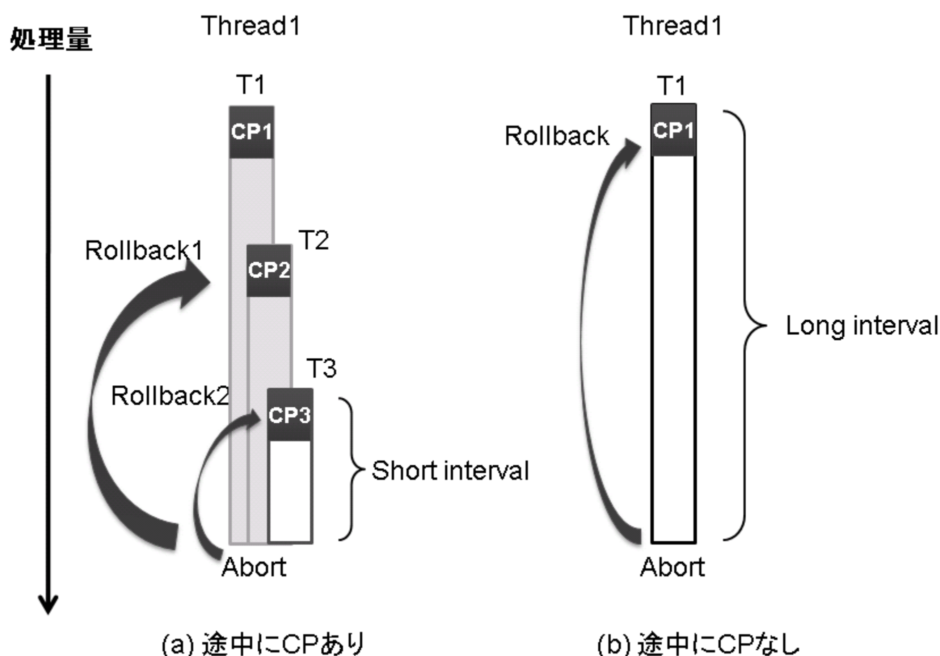

図 10: CP 間隔が広い場合の問題例

されてしまう場面が多くなってしまうことがある．したがって，チェックポイントの間隔が広くなる場合では，アボート位置からチェックポイント位置までのロールバックした命令を再実行するのにかかる**再実行コスト**が多くなってしまう可能性がある．

一方，トランザクションの間隔を短く設定してしまった場合でも問題がある．**図 11** 左のようにトランザクションの途中に複数のトランザクションを設定した場合では，図 11 右のように CP をまたいで同一アドレスへの書き込みが複数あると，書き込み前の値を複数ログに保存することになってしまう．この理由は，たとえばトランザクションを CP2 の状態に戻すには 0x100 番地の値を 20 にする必要がある．したがって，CP2 直後に退避したログを用いて値を書き戻す．このように，書き戻しに必要な値を退避するためには，チェックポイント後の初ストアをログへ保存する必要がある．この例のように，チェックポイントが複数ある場合では，ログへのアクセス回数が増え，また書き戻す量も増えてしまう可能性がある．以下，これらのコストを**チェックポイント作成によるコスト**と呼ぶ．

このように，プログラマによるトランザクションの範囲指定が最適でないと再実行コストやチェックポイント作成によるコストが大きくなってしまう．そこで，本研究ではプログラマがトランザクション内部で細かくトランザクションを設定しなくても，動的にチェックポイントを作成する手法を提案する．これによってトランザクション

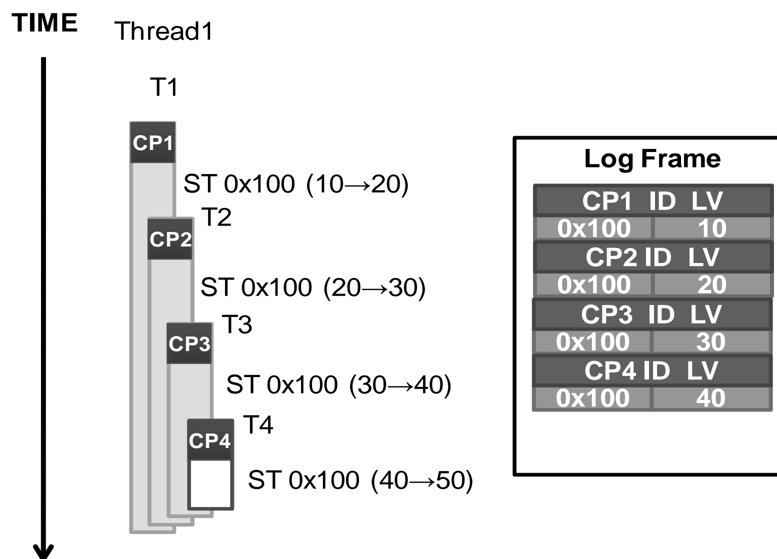

図 11: CP 間隔が短い場合の問題例

の内側に作成したチェックポイントから再実行する可能性が増え，再実行コストを削減することができる場合がある．

3.2 ロード/ストア回数に基づくチェックポイントの作成

競合はトランザクション内におけるロード命令とストア命令によって発生する．したがって，トランザクションの性質に合わせてチェックポイントを作成するためには，これらの命令を考慮することが望ましい．

そこで，本研究ではトランザクション内におけるロード/ストア命令の実行回数を用いた動的なチェックポイント作成手法を提案する．この手法では，ロード/ストア命令の実行回数を用いるため，これらの合計をカウントする．そして，その値に基づいて新たにチェックポイントを作成する．これにより，プログラマによって指定されたトランザクションが一つでも，トランザクションの途中から再開することができるようになる．

それでは，既存の LogTM と提案手法とで，トランザクションの実行中にどのような違いがあるのかを**図 12** に示す．図 12(a) の既存の LogTM，図 12(b) の提案手法ともに同一のトランザクションを実行し，提案手法の場合ではロード/ストア回数により 0x100 番地，0x200 番地へのストア後にそれぞれチェックポイントを作成したとする．また，それぞれの手法において，アボート位置はどちらも同じであるとする．既存の

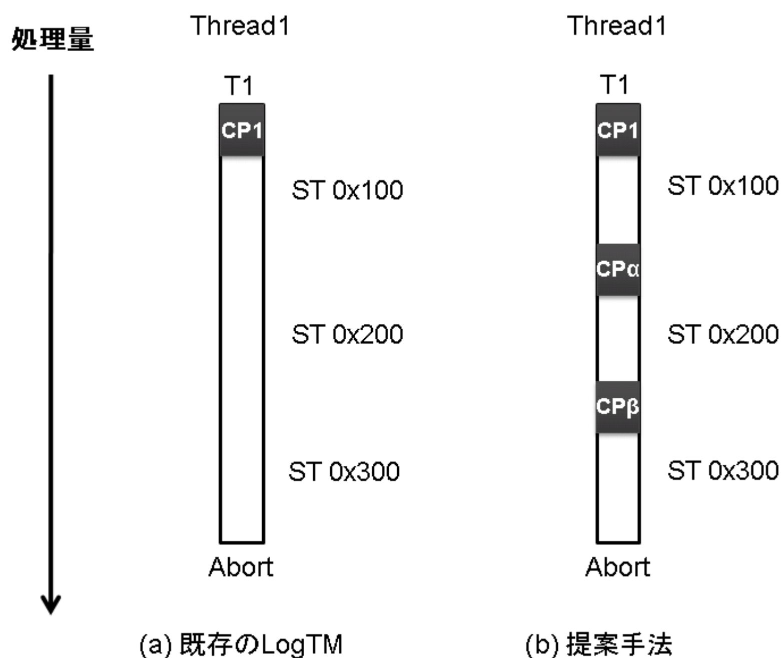

図 12: 既存の LogTM と提案手法の比較

LogTM ではどのアドレスに対して競合が発生しても CP1 から再実行する．しかし，提案手法では競合するアドレスによって再開位置が異なる．例えば，0x200 番地において競合が発生したとすると CPα から再実行し，0x300 番地ならば CPβ から再実行する．そのため，競合したアドレスへのアクセスよりも前にチェックポイントを作成することが重要である．ただし，0x100 番地において競合が発生した場合のようにどちらの手法も CP1 から再実行することもある．

3.3　チェックポイント間隔の決定手法

前節で紹介した手法では，一定のロード/ストア回数毎にチェックポイントを作成するので，トランザクションの長さによって問題が生じてくる．例えば，1000 ロード/ストアよりも後に競合が発生しやすいトランザクション内で，ロード/ストア回数が 100 回になるたびにチェックポイントを作成するとする．この場合では，最初の 9 個のチェックポイントはロールバック対象となる可能性が低く，無駄にチェックポイントを作成したことになる．したがって，チェックポイント作成によるコストが増加する．

一方，短いトランザクションの場合にはチェックポイントの数が少なくなってしまったり，そもそも作成できない可能性がある．例えば，100 ロード/ストアよりも前に競合が発生しやすいトランザクション内で，ロード/ストア回数が 1000 回になるたびに

チェックポイントを作成するとする．この場合では，どのチェックポイントもロールバック対象となる可能性が低いため，部分ロールバックが期待できない．また，トランザクション内で行われるロード/ストア命令数が 1000 回に満たなければトランザクション内にそもそもチェックポイントが作成されない．このような問題に対処するために，チェックポイント作成に用いるロード/ストア回数の設定をトランザクションの実行中に変動させる．

一般的に競合しやすいアドレスはトランザクションの前半部分に存在する．例えば，10 個のスレッドが同じトランザクションを実行するとする．そのうち 5 個のスレッドはトランザクションの 4 割まで実行し、残りのスレッド 5 個はトランザクションの 8 割まで実行しているとする．この場合では，全てのスレッドは既に 4 割まで実行を終えており，それより後から 8 割までの部分は 5 個のスレッドしか終えていない．つまり，プログラムの後半部分より，前半部分の方がより多くのスレッドが実行しており，そのことから，プログラムの前半部分で競合が発生しやすいことがわかる．したがって，プログラムの前半部分にチェックポイントを多く設定し，プログラムの後半部分はチェックポイントを少なく設定したい．そこで，これら 2 つの問題に対処するために，チェックポイントの作成間隔を次のような式で定義する．

$$LSC(k) = 2^{k+i} \qquad (k \geq 1 \quad i \geq 0) \tag{1}$$

式 (1) の LSC は次のチェックポイントまでのロード/ストアの実行回数を示している．つまり，k 回目のチェックポイントは，前のチェックポイントから $LSC(k)$ 回のロードストアが発生したときに作成される．また，i は最初に動的に作成するチェックポイントまでの間隔を設定するのに用いる．例えば，i の値を 2 として合計ロード/ストア回数 1000 回を含むトランザクションを実行する場合を考える．k の初期値は 1 なので，最初のチェックポイントまでは 8 個のロード/ストア命令，次のチェックポイントまではさらに 16 個のロード/ストア命令と増加していき，合計 6 個のチェックポイントが動的に作成される．このようにして，トランザクションの前半部分に多くチェックポイントを作成し，後半部分に作成するチェックポイント数を減らすことができる．

4 実装

本章では提案手法の実装について説明する．

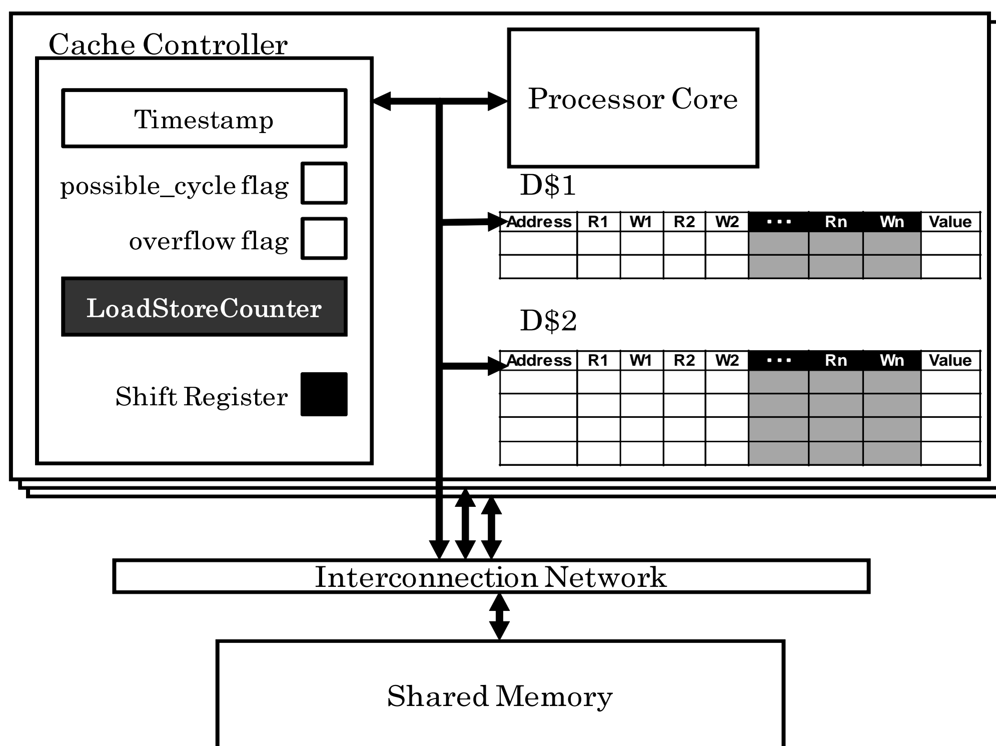

図 13: 拡張した LogTM の構成

4.1 ロード/ストアカウンタとチェックポイントの操作

3.2 で説明したように，チェックポイントを作成するためにロード命令とストア命令の数をカウントする．そのため，それらの命令を数えるカウンタを追加する．また，動的にチェックポイントの間隔を変更するためにシフトレジスタを追加した．以下，拡張した機構について説明する．

4.1.1 拡張した LogTM の構成

本提案手法ではトランザクション内で実行されたロード命令とストア命令をカウントする LoadStoreCounter，ロールバック先のチェックポイントを選択するために，D$1，D$2 上にある R1，R2 や W1，W2 に加えてさらに Rn，Wn までの合計 n 個の read セット及び write セットを加える．また，3.3 で説明した $LSC(k)$ の値を格納する Shift Register を追加した．

4.1.2 ロード/ストアカウンタ

ロード/ストアカウンタは，トランザクション内でロード命令及びストア命令が実行されるとその数をカウントする．カウント数が特定の値になると動的にチェックポイントが作成される．

一方，カウンタの値をリセットするタイミングはチェックポイントの作成時及びトランザクションの再実行時である．提案手法によりチェックポイントを作成した場合は，カウンタの値をリセットする．また，トランザクション開始時には，カウンタの値が指定したロード/ストア回数に満たなくてもチェックポイントが作成される．ここで，カウンタの値をリセットしなければ次のチェックポイントまでの間隔が短くなってしまうため，カウンタの値をリセットする．

トランザクションの再実行時にはカウンタの値をそのチェックポイントを作成した時のロード/ストア回数に戻す必要がある．しかし，そのチェックポイントを作成した時にカウンタをリセットしているため，再実行するときにカウンタの値をリセットするだけでチェックポイント作成時と同じロード/ストア回数に戻すことができる．

また，トランザクションの終了時にはカウンタの値をリセットする必要はない．なぜなら，次のトランザクション開始時にカウンタの値をリセットするからである．ロード/ストアカウンタは以上のような流れをくりかえすことで実現することができる．

4.2　動的チェックポイントの削除条件

既存の LogTM では，コミットまで実行が進むとチェックポイントは削除される．しかし，提案手法により作成されたチェックポイントは，対応するコミットが存在しないため削除されない．したがって，動的にコミット操作を追加し，チェックポイントを削除しなければならない．まず，チェックポイントにはプログラム中で規定されたトランザクション開始時に設定されたものと，提案手法により作成されるものの 2 種類が存在する．LogTM ではトランザクションに ID を付けることになっており，この ID は 2.4 節で説明したように，ログのヘッダ部に保持される．そこで，提案手法ではこの ID を用いてチェックポイントの削除を行う．提案手法により作成されるチェックポイントには直前のトランザクション開始時のチェックポイントと同じ ID を割り当てるとする．

それでは，**図 14** を用いて具体的に説明する．図 14(a) は，トランザクションの開始時にのみチェックポイントが作成される既存モデルの例である．また，図 14(b) では，さらに提案手法によりチェックポイントの CPα，CPβ が作成される．トランザクション開始時のチェックポイントの ID をそれぞれ 1，2 とすると，提案手法により作成される CPα の ID は 1，CPβ の ID は 2 となる．図 14 においてトランザクションのコミット位置である Commit ID2 に到達すると，ID2 に対応するチェックポイントを削除する．図 14(a) の場合ではこの ID2 が割り当てられた CP2 のみが削除される．しかし，

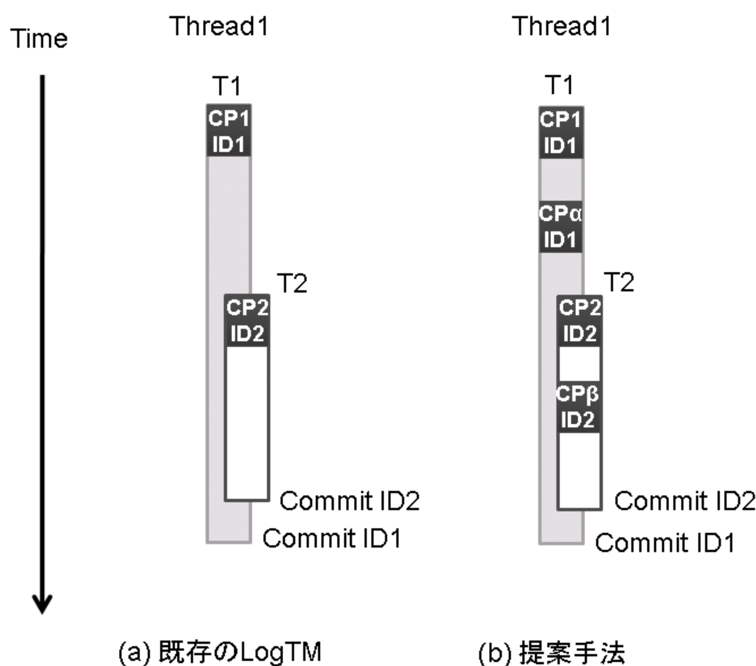

図 14: チェックポイントの ID 割り当て

図 14(b) の場合では対応するチェックポイントが複数存在するため，内側のトランザクションの開始点である CP2 と提案手法により作成された CPβ の合計 2 つのチェックポイントが削除される．次のコミット時も同様に処理をするため，ID に対応するチェックポイントである CP1 と CPα が削除される．以上のようにして動的に作成したチェックポイントを削除する．

4.3 チェックポイント間隔の動的変更

チェックポイントの作成条件をトランザクションの実行中に変更するために 3.3 節で説明した式を用いて実現する．この提案手法では，4.1.2 項で説明したロード/ストアカウンタの値が，式 (1) で表される $LSC(k)$ と同じ値になるとチェックポイントを作成し，k の値をインクリメントする．k のインクリメントは LSC の値を 2 倍にする計算なので，ビットシフトにより実現できる．したがって，LSC の値はシフトレジスタに格納する．例えば，k の値が 1，i の値が 0 の場合では LSC は 2 となるので，シフトレジスタの下位 2 ビット目がセットされていれば良い．また，それ以後の LSC の値は 2 倍ずつ増加していくため，シフトレジスタに格納されている値を 1 ビット左シフトすることにより表現可能である．これにより，シフトレジスタを用いるだけでチェックポイント作成間隔の変更に対応することが可能となり，少ないハードウェアコストで

提案手法を実現できる．また，トランザクションの開始位置では，ロード/ストアカウンタの値が $LSC(k)$ の値に満たなくてもチェックポイントを作成するので，先ほどと同様にシフトレジスタを 1 ビット左シフトする．一方，コミット時は k の値は変更しない．

しかし，アボートが発生した場合には，シフトレジスタを右シフトすることにより，LSC の値をロールバック先のチェックポイントを作成した時と同じ値に戻す必要がある．そこで，アボート時のトランザクションレベルと再実行開始位置でのトランザクションレベルから，右シフトしなければならないビット数 RSN を求める．アボート時のトランザクションレベルを $t2$，再実行開始位置のトランザクションレベル $t1$ とすると，次のような式で RSN を表現することができる．

$$RSN = t2 - t1 \tag{2}$$

それぞれのトランザクションレベルに対して，式 (1) の i は同じ値である．よって，i を考慮する必要は無く，トランザクションレベルのみを用いて LSC を適切な値に戻すことができる．

例えば，アボート時のトランザクションレベルを 4 とし，再実行開始位置のトランザクションレベルを 2 とする．ここで，式 (2) より RSN の値は 2 となるので，2 ビットの右シフトにより，適切な LSC の値に戻すことができる．

5　評価

5.1　評価環境

前章までで述べた拡張を既存の LogTM に実装し，シミュレーションによる評価を行った．評価にはフルシステムシミュレータである Simics[6] と GEMS-2.1.1[7] を用いた．

Simics は機能シミュレーション，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行う．プロセッサは 32 コアの SPARC V9 とし，各コアは単命令・インオーダ発行を行う．また，OS は Solaris10 とした．**表 1** に詳細なシミュレーション環境を示す．

評価対象のプログラムとしては GEMS 付属ベンチマークプログラムである Contention，Prioque，Slist を用い，それぞれのプログラムを 2，4，8 スレッドで実行した．それぞれの入力を**表 2** に示す．

なお，本実験はフルシステムシミュレータ上においてマルチスレッドを用いた動作

表 1: システムモデルパラメータ

Processor	32 cores
周波数	4 GHz
	single-issue
	in-order
	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	200 cycles
Interconnect network latency	3 cycles

表 2: ベンチマークプログラムとその入力

Contention	config 1
Prioque	8192ops
Slist	500ops 64len

のシミュレーションを行っているため，実行するたびに結果が変動する．したがって，各評価対象につき試行を 5 回行い，得られたサイクル数の平均値を比較する．

5.2 評価結果

図 15，**図 16**，**図 17** に Contention，Prioque，Slist の評価結果を示す．また，それぞれの評価結果のグラフは左から順に

(F) 必ずトランザクションの開始点までロールバック

(P) トランザクションの途中から再実行可能

(C1) 2^{k+i} を用いた提案手法で $i = 2$ として CP 作成開始

(C2) 2^{k+i} を用いた提案手法で $i = 3$ として CP 作成開始

が要した実行サイクル数を表している．なお，モデル (F) の実行サイクル数を 1 とし

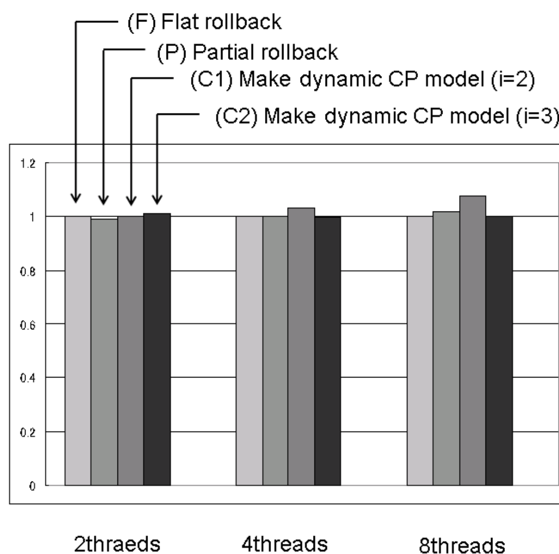

図 15: 評価結果 (contention)

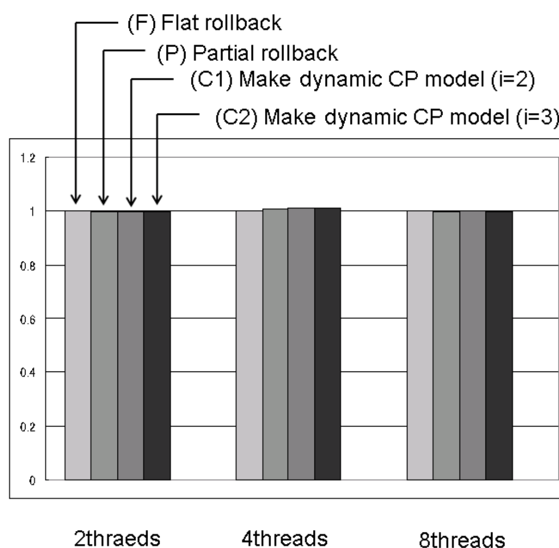

図 16: 評価結果 (prioque)

て正規化した．

それぞれのプログラムにおいて，スレッド数を 2 として実行した場合ではアボートがほとんど発生しておらず，実行サイクル数はほぼ変わっていない．一方で，スレッド数が多い場合はアボートが多く発生したが，ロールバック先のトランザクションレベルは低くなりやすい傾向が見られた．したがって，コストを削減できる回数は増えたが，1 回に削減できるコストの量は減少している．Prioque は，提案手法によって作成

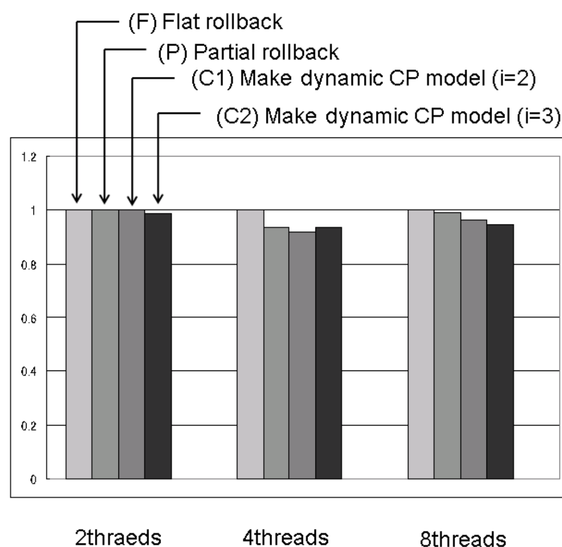

図 17: 評価結果 (slist)

されたチェックポイントへロールバックする割合が低く既存モデルと提案モデルでのサイクル数の差は少ない結果となった．一方 Slist も同様の傾向が見られたが，トランザクションが長いために (C1) モデルでは (P) モデルより 4 スレッドで 4.43%，8 スレッドで 2.53% の実行サイクル数が削減できた．また Contention では，(C1) モデルによる効果が少ないため，(P) モデルと比べて 4 スレッドでで 2.89%，8 スレッドで 5.84%実行サイクル数が増加した．

5.3 考察

まず，Contention はスレッド数が多い程競合が顕著に見られ，アボート処理の遅い LogTM ではスレッド数が多い程実行サイクル数が増加した．また，スレッド数 4，8 では提案手法は共にアボート数が増加したが，モデル (C2) では実行サイクル数の削減が見られた．なお，実行サイクル数があまり減少していない理由は，作成したチェックポイントによって書き戻しの量が増え，ロールバック先に選ばれにくい無駄なチェックポイントを作成してしまったためであると考えられる．(C1) のサイクル数が (C2) よりも増加している理由は，ロールバック先の対象に選択されないチェックポイントを余分に 1 つ作成しているためである．

次に，Prioque で既存モデルと提案モデルで変化が少ないのは，削減できた命令はプログラム前半のロード命令のため，提案モデルによるロード命令の再実行コスト削減と，チェックポイント作成によるコストがつり合っているためだと考えられる．

表 3: アボート回数とロールバック (4threads)

	Flat	Partial	提案 (k=3)	PCPR	提案 (k=4)	PCPR
Contention	1192	1241	2203	52.76%	1868.4	53.10%
Prioque	337.6	350.8	339.7	1.15%	326.4	0.80%
Slist	594.5	575.5	573.25	0.52%	583.25	0%

表 4: アボート回数とロールバック (8threads)

	Flat	Partial	提案 (k=3)	PCPR	提案 (k=4)	PCPR
Contention	4460	48025.6	7172.5	52.06%	7971	51.95%
Prioque	1860.4	1772.8	1704.7	0.64%	1673.4	0.93%
Slist	1152	1194.75	1190.75	0.59%	2029.5	0.22%

一方，Slist ではトランザクションが入れ子になっており，この内側のトランザクション開始点に戻る割合はスレッド数が少ない程多かった．そのためスレッド数が少ない場合において，モデル (P) のサイクル数がモデル (F) より多く減少している．一方，提案手法によるチェックポイントへロールバックする場合もあったが，その割合が少ないため，チェックポイント作成によるコストの影響の方が勝ってしまったと考えられる．

次に，モデル (F)，(P)，(C1)，(C2) それぞれの合計アボート回数と，提案手法によって作成されたチェックポイントへロールバックする割合を**表 3** と**表 4** に示す．表 3 は，スレッド数を 4 として実行した結果で，表 4 では，スレッド数を 8 として実行した結果を示している．なお，スレッド数を 2 として実行した場合では，アボート回数が極めて少なかったため，評価の対象外とした．また，これらの表にある PCPR とは，全再実行回数のうち，提案手法によって作成されたチェックポイントから再実行した割合を示している．

Contention において，既存手法に比べて提案手法のアボート数が増加した理由は，提案手法によるチェックポイントから再実行する割合が約 52%となっており，アボート時の書き戻しにかかる時間が減少したため，トランザクションの実行している時間の割合が増加し，再びアボートが発生する可能性が増えたからだと考えられる．一方，Prioque と Slist では削減できた書き戻しコストが少ないため，アボート数の増加にはつながっていない．しかし，PCPR が低いため，サイクル数の減少にほとんど寄与していない．

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，再実行コストを削減する手法を提案した．拡張したLogTM ではチェックポイント作成条件にロード/ストア回数を用いて，トランザクションの途中にチェックポイントを作成する．また，トランザクションの長さに合わせてチェックポイント間隔を動的に変動させる手法を提案した．提案手法の有効性を確認するため，付属ベンチマークプログラムであるContention，Priorityqueue，Slist を用いて評価を行った．その結果，提案手法(C1)，(C2) が有効である場合と有効でない場合とが確認できた．有効となった要因は，アボート時の書き戻しコストや再実行コストが減少したことである．しかし，チェックポイント作成によるコストのために性能が悪化してしまう場合が多く見られた．したがって，本研究の今後の課題として，ログの書き戻しにかかるコストの削減や，チェックポイントの位置をより競合しやすい位置の直前に作成し，再実行コストを下げることが考えられる．また，部分ロールバックをより効率的なものとするため，アボート対象の選択方法を変更することも考えられる．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩助教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・津邑研究室およびの齋藤研究室の方々に深く感謝致します．特に，浅井宏樹氏，池谷友基氏，今井満寿巳氏には研究を進めるにあたって何度も相談にのっていただき，そのたびに貴重なご意見を頂きました．ここに深く感謝致します．

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, *Proc. of 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, *Proc. of 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, *Proc. of 13th Annual Int'l. Symp. on Computer Architecture (ISCA'86)*, pp. 414–423 (1986).

[4] 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).

[5] Rajwar, R. and Goodman, J. R.: Transactional Lock-Free Execution of Lock-Based Programs, *Proc of 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 5–17 (2002).

[6] 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).

[7] 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, *AMC SIGARCH Computer Architecture News*, Vol. 33, No. 4, pp. 92–99 (2005).