

Read-after-Read アクセスを制御する ハードウェアトランザクショナルメモリ

橋本 高志良¹ 鈴木 大輝¹ 堀場 匠一朗¹ 江藤 正通¹ 津邑 公暁^{1,a)} 松尾 啓志¹

概要:

マルチコア環境では、一般的にロックを用いて共有変数へのアクセスを調停する。しかし、ロックには並列性の低下やデッドロックの発生などの問題があるため、これに代わる並行性制御機構としてトランザクショナル・メモリが提案されている。この機構においては、アクセス競合が発生しない限りトランザクションが投機的に実行されるため、一般にロックよりも並列性が向上する。しかし、Read-after-Read アクセスが発生した際に投機実行を継続した場合、その後発生するストールが完全に無駄となる場合がある。本稿では、このような問題を引き起こす Read-after-Read アクセスを検出し、それに関与するトランザクションを敢えて逐次実行することで、全体性能を向上させる手法を提案する。シミュレーションによる評価の結果、提案手法により最大 66.9%、平均 22.6%の高速化を確認した。

1. はじめに

マルチコア環境において一般的な共有メモリ型並列プログラミングでは、共有リソースへのアクセスを調停する機構として、一般にロックが用いられてきた。しかしロックを用いた場合、ロック操作のオーバヘッドに伴う並列性の低下や、デッドロックの発生などの問題が起こりうる。さらに、プログラムごとに適切なロック粒度を設定するのは困難であるため、この機構はプログラマにとって必ずしも利用し易いものではない。そこで、ロックを用いない並行性制御機構としてトランザクショナル・メモリ (Transactional Memory: TM) [1] が提案されている。

TM では、従来ロックで保護されていたクリティカルセクションをトランザクションとして定義し、共有リソースへのアクセスにおいて競合が発生しない限り、投機的に実行を進めるため、ロックを用いる場合よりも並列性が向上する。なお、トランザクションの実行中においては、その実行が投機的であるがゆえ、共有リソースに対する更新の際には更新前の値を保持しておく必要がある (バージョン管理)。また、トランザクションを実行するスレッド間において、共有リソースに対する競合が発生していないかを常に検査する必要がある (競合検出)。TM のハードウェア実装であるハードウェア・トランザクショナル・メモリ (Hardware Transactional Memory: HTM) では、このバージョン管理および競合検出のための機構をハード

ウェアで実現することで、これらの処理を高速化している。

さて、上述のとおり HTM では競合が発生しない限りトランザクションが投機的に並列実行される。ここで、あるトランザクションが Read アクセス済である変数に対し、他のトランザクションが Read アクセスしようとした場合、すなわち Read-after-Read (RaR) アクセスが発生した場合、競合とはならず、投機実行は継続される。しかし、それらのトランザクションの一方が結果的にアボートした場合、その過程において発生したストールは完全に無駄となる。我々はこれが HTM の全体性能を大きく低下させてしまう場合があることを発見した。そこで本稿では、このような問題を起こし得る RaR アクセスを検出し、そのアクセスに関与したトランザクションを敢えて逐次実行することで、HTM の性能を向上させる手法を提案する。

2. 関連研究

アボートしたトランザクションを途中から再実行することで、その再実行コストを抑える部分ロールバック [2], [3] の研究や、バージョン管理や競合検出の方式を動的に変更する研究 [4], [5] など数多くの HTM に関する研究が行われてきた。特にスレッドスケジューリングに関しては、これまで主に 2 つの方向性から改良手法が提案されてきた。

競合の発生を抑制するという観点から行われた研究として、次の 3 つの手法が挙げられる。まず、Yoo ら [6] は HTM に Adaptive Transaction Scheduling (ATS) と呼ばれるシステムを実装し、競合の頻発によって並列性が著しく低下するアプリケーションの実行を高速化する手法を提案した。一方で、Geoffrey ら [7] は複数のトランザクシ

¹ 名古屋工業大学
Nagoya Institute of Technology, Nagoya, Aichi, 466-8555, Japan

a) tsumura@nitech.ac.jp

ン内でアクセスされるアドレスの局所性を Similarity と定義し、これが一定の閾値を超えた場合に、当該トランザクションを逐次実行する手法を提案した。また、Akpinar [8] は HTM の性能を低下させるような競合パターンに対する、様々な競合解決手法を提案した。

もう一方の方向性からの改良として、Gaona ら [9] は消費電力を抑えるという観点から、複数のトランザクション間で競合が発生した場合に、その競合に関与したトランザクションに実行優先度を設定し、それらを逐次実行することで消費エネルギーを削減する手法を提案した。

以上に述べた手法は、いずれもアボートや競合の発生回数などの情報に基づいてスレッドの振る舞いを決定しており、それらのスレッドが共有リソースにアクセスする順序を考慮していない。そのため、HTM の性能を低下させる競合パターンが根本的に解決されておらず、目立った性能向上を得ることはできていない。一方本稿では、共有リソースへのアクセス順序に着目して、上述したスケジューリング手法では解決できていなかった競合パターンの効果的な解決を図る。

3. 競合アドレス情報に基づく並列度の低減

本章では、既存の HTM における問題点と、それを解決する手法について述べる。

3.1 Read-after-Read アクセスに起因する問題

一般に、共有変数への Read アクセスは、その後に Write アクセスを伴う場合が多く見られる。具体的には Test-and-Set のような操作を実現する場合や、演算結果をある変数にアキュムレートしていく場合などがこれにあたる。このような、ある共有変数に対し Write アクセスに先立って Read アクセスが行われるようなトランザクション処理が、複数のスレッドにより並列に実行される場合、両スレッドの Read アクセスが競合とならず許可されたとしても、その後実行される Write アクセスにより結局競合が発生してしまうことになり、これが性能低下を引き起こし得る。

図 1 は、上記の処理を含むトランザクション $Tx.X$ を、2つのスレッド $Thread1$ および $Thread2$ が並列に実行する様子を示している。まず、両スレッドが load A を実行した後、 $Thread2$ が store A を実行しようとする場合、競合の発生により、 $Thread2$ は自身の $Tx.X$ をストールする (時刻 $t1$)。その後、 $Thread1$ が store A を実行しようとする場合 ($t2$)、 $Thread2$ は既に当該アドレスにアクセス済であるため競合を検出し、 $Thread1$ へ NACK を返信する。この時、 $Thread1$ は自身よりも早くトランザクションを開始したスレッドから NACK を受信するため、 $Tx.X$ をアボートする ($t3$)。このアボートにより、 $Thread2$ は $Tx.X$ を再開できるが、この間に $Thread1$ の実行は一切進行しておらず、 $Thread2$ のストールは完全に無駄であったことになる。このようなストールを **Futile Stall** と呼び、HTM のスループットを低下させる大きな要因となる。

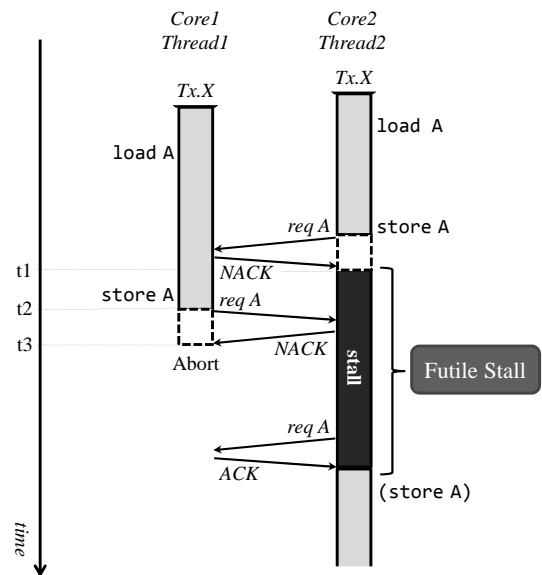


図 1 Read-after-Read アクセスに起因する Futile Stall

3.2 Futile Stall 抑制手法

本節では、Futile Stall の発生を抑制し、性能低下を防ぐ手法を提案する。

3.2.1 基本動作

Futile Stall が発生する要因として、あるアドレスに対して複数のスレッドが、Write アクセスに先んじて Read アクセスすることで、両スレッドが当該アドレスにアクセス済になってしまうことが考えられる。そこで、Read → Write の順序でアクセスされるアドレスに対する Read アクセスの際に、それが RaR アクセスであるか否かを検出する。そして RaR アクセスであった場合、即時には Read アクセスを許可せず待機させ、Read アクセス済のスレッドが実行トランザクションをコミットした時点で、それらのアクセスを順次許可する手法を提案する。これにより RaR アクセスに関与したトランザクションの実行を逐次化することで Futile Stall を回避する。

ここで図 2 に、提案する Futile Stall 抑制手法を用いた場合の動作を示す。この例では、3つのスレッド ($Thread1 \sim 3$) がそれぞれ同一のトランザクション ($Tx.X$) を投機実行している。まず、 $Thread2$ が load A を実行した後、 $Thread1$ と $Thread3$ が load A を実行しようとした場合 (時刻 $t1, t2$)、 $Thread2$ は RaR アクセスを検出し、それぞれのスレッドに、実行を待機させる通知である Wait リクエストを送信する。Wait リクエストはコヒーレンスプロトコルを拡張する形で新たに定義する。この Wait リクエストの受信により ($t3, t4$)、 $Thread1$ と $Thread3$ の実行は待機させられるため、 $Thread2$ はアドレス A に Write アクセスしたとしても、図 1 の場合とは異なり、これらのスレッドと競合することなく $Tx.X$ の実行を進めることができ、Futile Stall による無駄なサイクルを削減できる。

3.2.2 待機スレッドの再開順序制御

前項で述べた手法により $Thread2$ の Futile Stall は回避できるが、 $Thread2$ は実行トランザクションをコミットし

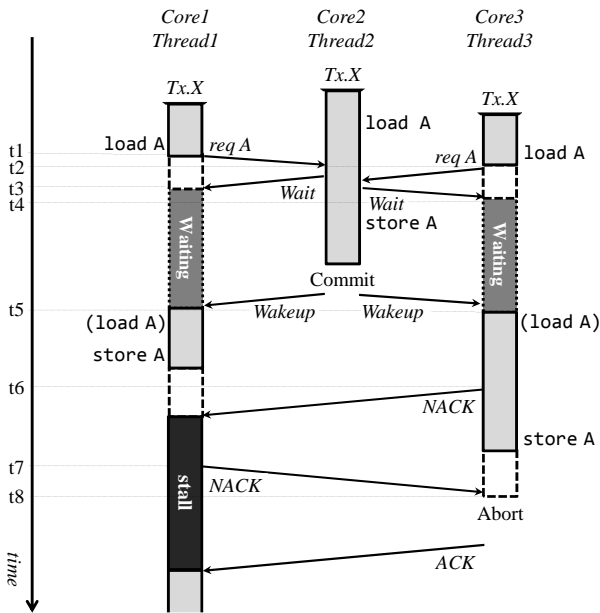


図 2 RaR アクセスの検出による Futile Stall の抑制

た際、*Thread1* と *Thread3* の待機状態を解除する必要がある。このため、*Wakeup* メッセージを新たに定義し、これを送信することで待機スレッドを再開させる (図 2, t_5)。しかし、この例のように待機スレッドが複数存在する場合、単純に *Thread1* および *Thread3* に同時に *Wakeup* メッセージを送信し、これらを一齐に再開させたのでは、*Thread1* と *Thread3* の間で再度競合が発生してしまう (t_6, t_7)。なお、簡略化のために図 2 において、時刻 t_5 以降のアドレス A に対するリクエストの表記は省略している。その後、発生した競合により *Thread3* が *Tx.X* を結果としてアボートするため (t_8)、*Thread1* のストールが無駄となってしまう。

これを解決するため、待機スレッドの再開順序を制御する手法を併せて提案する。これは待機させる側のスレッドが、結果的に待機させられたスレッドからの Read リクエストを受信した順に記憶しておき、実行トランザクションのコミット時にその順序で *Wakeup* していくことで実現する。図 2 の例の場合、*Thread1* と *Thread3* を待機させた *Thread2* が実行トランザクションをコミットした際、記憶した順序にしたがって待機スレッドを再開させる。図 2 では、*Thread3* より先に *Thread1* が Read アクセスを試みているため、*Thread2* は最初に *Thread1* の実行を再開させる。実行を再開した *Thread1* は、実行トランザクションをコミットした際、再開順序を制御する *Thread2* へコミットしたことを伝える。*Thread2* は *Thread1* のコミットを検知すると、続けて *Thread3* の実行を再開させる。以上のように動作させることで、RaR アクセスを検出した *Thread2* による待機スレッドの再開順序制御を実現する。

4. 実装

本章では提案手法を実現するために拡張したハードウェアと、具体的な動作モデルについて述べる。

4.1 拡張ハードウェア構成

提案手法を実現するため、以下のハードウェアを各コアに追加する。

Register for RaR addresses (RaR addr.) :

各スレッドにおいて Read → Write の順序でアクセスされたアドレスを記憶するレジスタ。

Queue for order of resumption (O-que.) :

RaR アクセスを検出することで、他のスレッドを待機させたスレッドが再開順序を制御するために用いるキュー。これには、RaR addr. に登録されたアドレスに対して、Read アクセスを試みたスレッドを実行するコア番号と、そのアクセス順序が記憶される。

Register for resumption manager (R-res.) :

RaR アクセスの検出によって実行を待機させられたスレッドが用いるレジスタ。これには再開順序を制御しているスレッドを実行するコア番号が記憶され、待機スレッドは実行を再開して実行トランザクションをコミットした際、記憶されているコア番号に対応するスレッドへコミットしたことを伝える。

各スレッドは、Read → Write の順序でアクセスしたアドレスを、RaR addr. に保持する。これはアドレスを複数記憶するようにも構成できる。そして、各スレッドは他スレッドから Read アクセスのためのリクエストを受信した際に、RaR addr. を参照して RaR アクセスを検出すべきアドレスに対する Read アクセスか否かを判定する。さらに、待機スレッドを順に再開させるために O-que. を追加する。RaR アクセスを検出して他のスレッドを待機させたスレッドは、実行トランザクションをコミットもしくはアボートした場合に O-que. に記憶されたアクセス順序に基づいて再開順序を制御する。また、再開順序を制御するスレッドは、実行を再開させたスレッドがトランザクションをコミットしたことを確認後、次の待機スレッドを再開させる必要がある。そのため、待機スレッドは再開順序を制御しているスレッドを実行するコア番号を R-res. に記憶し、実行トランザクションをコミットした際、R-res. に記憶したコア番号に対応するスレッドに対してコミットしたことを伝える。

4.2 Read-after-Read アクセス検出機構

本節では RaR アクセスを検出する動作モデルについて述べる。

4.2.1 RaR addr. へのアドレス登録

3つのスレッド (*Thread1* ~ *3*) がそれぞれ同一のトランザクション (*Tx.X*) を投機実行している図 3 を例に、追加した RaR addr. へのアドレス登録の動作を述べる。まず、各スレッドが load A を実行した後、*Thread1* が store A を実行しようとする場合 (時刻 t_1)、Write-after-Read (WaR) 競合の発生により、*Thread2* と *Thread3* から NACK が返信されるため、*Thread1* は自身の *Tx.X* をストールする (t_2)。続いて、*Thread2* と *Thread3* がそれぞれ store A を

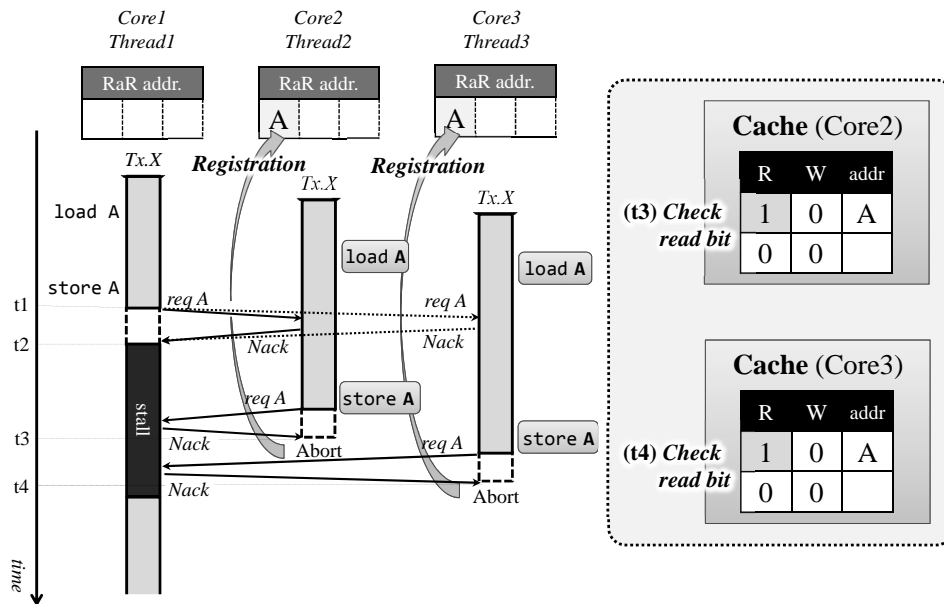


図 3 RaR アクセスを検出すべきアドレスの検知と RaR addr. への登録

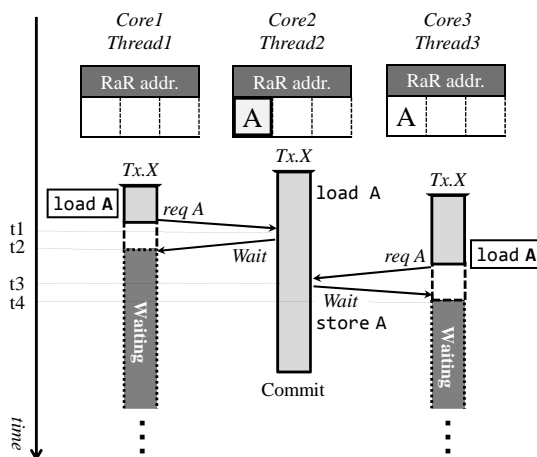


図 4 RaR addr. を利用した RaR アクセスの検出

実行しようとするが、Thread1 との間でそれぞれ WaR 競合が発生するため、両スレッドは自身の実行中トランザクションのアボートを試みる。この時、Thread2 と Thread3 はアクセスしようとしていたアドレス A における自身の R ビットをチェックする (t3, t4)。この R ビットは、既存の HTM において競合を検出するために各キャッシュライン毎に付加されているものであり、そのラインのアドレスに対する Read アクセスが発生した場合にセットされる。当該アドレスの R ビットがセットされている場合、Thread2 と Thread3 は、自身が Write アクセスに先立ってアドレス A に Read アクセスしたことが分かるため、アドレス A を自身の RaR addr. に登録する。

4.2.2 RaR addr. の利用

4.2.1 項で述べた方法で RaR addr. に登録されたアドレスを利用して RaR アクセスを検出する動作を図 4 に示す。はじめに、3 つのスレッド (Thread1~3) は同一のトラン

ザクション (Tx.X) を実行し、Read アクセスのリクエストを受信するたびに RaR addr. を参照することとする。

図 4 の例では、既に Thread2 の RaR addr. にアドレス A が登録されているとする。まず、Thread2 が load A を実行後、Thread1 が load A の実行を試みるとする。この時、Thread1 は Thread2 へ、A に対するアクセスリクエストである req A を送信する (t1)。この req A を受信した Thread2 は、自身の RaR addr. を参照し、アドレス A が登録済みかどうかを確認する。Thread2 の RaR addr. には当該アドレス A が既に登録されているため、Thread2 はこの Read 要求が、自身が以前に Read → Write の順序でアクセスしたアドレス A に対する Read 要求であると分かる。したがって、Thread2 は RaR アクセスを検出し、Thread1 へ Wait リクエストを送信する。この Wait リクエストを受信した Thread1 は、Thread2 から Wakeup メッセージを受信するまで実行を待機する (t2)。その後、Thread3 が load A を実行しようとする場合も同様に (t3)、Thread3 は RaR アクセスを検出した Thread2 から返信される Wait リクエストを受信した後、実行を待機する (t4)。

4.2.3 RaR addr. のハードウェアコスト

ここで、RaR addr. のハードウェアコストについて検討する。4.1 節で示したように、RaR addr. には Read → Write の順序でアクセスされたアドレスが登録される。しかし、1 つのプログラム中において Read → Write の順序でアクセスされるアドレスを全て記憶できるだけの容量を準備することは現実的ではない。したがって、RaR addr. に登録できるアドレス数を最大 N としてコストを抑える。登録アドレス数 N を 1, 2, または 4 と設定した場合、それぞれコアあたり 64bit, 128bit, 256bit のコストで実現可能であり、プロセッサ全体でも、コア数を 32 とするとそれぞれ 256byte, 512byte, 1Kbyte と少量で実現できる。なお、登録可能なアドレス数を制限した場合、登録アドレス

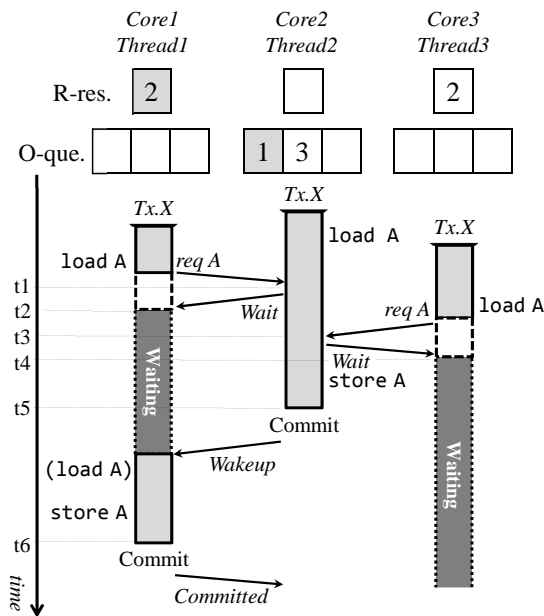


図 5 O-que. と R-res. の利用

の管理はいくつかの選択肢をとり得るが、本稿では実装を単純化するため、単純な FIFO を採用する。

この RaR addr. への登録アドレス数を増加させた場合、RaR アクセスを検出すべきアドレスをより多く記憶できるため性能が向上する可能性があるが、ハードウェアコストとのバランスを考える必要がある。そこで、登録数を増加させた場合の性能向上率とハードウェアコストのバランスを、実現性の観点から 5 章で考察する。

4.3 再開順序制御の実現

本節では、4.2 節で述べた方法によって他スレッドを待機させたスレッドが、待機スレッドの再開順序を制御する動作を図 5 に示す。この例は、図 4 の例で Thread2 が RaR アクセスを検出した後、Thread1 と Thread3 の再開順序を制御する動作例である。この例において、RaR アクセスを検出した Thread2 は、Read アクセスを試みた Thread1 を自身が待機させたスレッドと判断し、自身の O-que. に Thread1 を実行するコア番号を格納する。RaR アクセスの検出により実行を待機する Thread1 は、Thread2 を再開順序制御するスレッドだと判断し、自身の R-res. に Thread2 を実行するコア番号を格納する。その後、Thread3 が load Aを試みる場合も RaR アクセスが検出されるため、Thread2 は自身の O-que. に Thread3 を実行するコア番号を格納する。そして、Thread3 は R-res. に Thread2 を実行するコア番号を格納する。

次に O-que. と R-res. に格納したスレッド番号を利用して、待機スレッドの再開順序を制御する。まず Thread2 は Tx.X をコミットした際 (t5)、自身の O-que. に格納されている番号をチェックする。この時、Thread2 の O-que. にはコア番号 1, 3 が格納されており、Thread2 はこの O-que から先頭の値を取り出す。この例ではこれが 1 であること

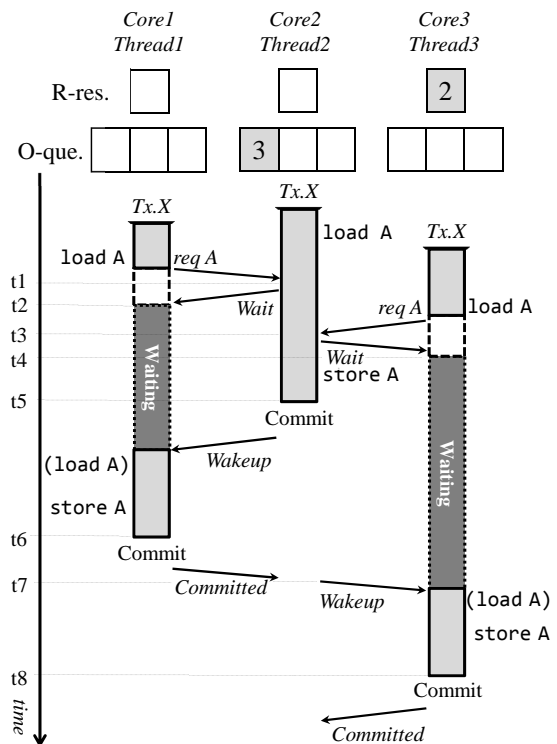


図 6 再開順序制御によるトランザクションの逐次実行

から、最初に再開させるべきスレッドは Core1 の実行するスレッドであると判断し、この Thread1 に対して Wakeup メッセージを送信する。この Wakeup メッセージを受信した Thread1 は Tx.X の実行を再開後にコミットに至る。Tx.X をコミットした Thread1 は、自身の R-res. に格納されているコア番号 2 を取り出し、Committed 通知を送信することで、Tx.X をコミットしたことを Thread2 に伝える (t6)。

このようにして Committed 通知を受信した Thread2 は、図 6 に示すように再び自身の O-que. をチェックし (時刻 t7)、コア番号 3 を取り出すことになるため、Thread3 に対して Wakeup メッセージを送信する。この Wakeup メッセージを受信した Thread3 は Thread1 の場合と同様に、実行を再開して Tx.X をコミットする。Thread3 は Tx.X をコミットした後、R-res. からコア番号を取り出し、Thread2 に対して Committed 通知を送信する (t8)。この Committed 通知を受信した Thread2 は、再度自身の O-que. をチェックする。この時、O-que. にはコア番号が格納されていないため、Thread2 は自身が待機させたスレッドの実行を全て再開させたと判断し、再開順序制御を終了する。

ここで、O-que. と R-res. のコストについて検討する。これらのエン트리にはコア番号が格納されるため、32 コア構成のプロセッサの場合 1 エン트리あたり 4bit 必要となる。また、O-que. には、最大で自コアを除く全てのコア番号を記憶する必要があるため、4bit × 31 の記憶容量が必要となる。以上より、必要となる総記憶容量は、4bit × 32 × 32 = 512bytes と少量である。

表 1 シミュレータ諸元
Table 1 Simulation parameters.

Processor	SPARC V9
#cores	32 cores
clock	1 GHz
issue width	single
issue order	in-order
non-memory IPC	1
D1 cache	32 KBytes
ways	4 ways
latency	3 cycles
D2 cache	8 MBytes
ways	8 ways
latency	34 cycles
Memory	8 GBytes
latency	500 cycles
Interconnect network latency	14 cycles

表 2 各ベンチマークにおける削減サイクル数

		GEMS	SPLASH-2	STAMP	All
(R_1)	平均	29.2%	19.1%	4.9%	22.6%
	最大	66.9%	39.9%	9.3%	66.9%
(R_2)	平均	29.3%	19.9%	5.2%	23.0%
	最大	66.9%	41.5%	9.9%	66.9%
(R_4)	平均	29.5%	19.9%	5.0%	23.1%
	最大	66.9%	41.1%	9.3%	66.9%
(R_∞)	平均	29.8%	22.4%	4.7%	24.0%
	最大	66.9%	40.9%	8.8%	66.9%

5. 評価

本章では、提案手法の速度性能をシミュレーションにより評価し、得られた評価結果から考察を行う。

5.1 評価環境

これまで述べた拡張を HTM の一実装である LogTM[10] に実装し、シミュレーションによる評価を行った。評価にはトランザクショナルメモリの研究で広く用いられている Simics[11] 3.0.31 と GEMS[12] 2.1.1 の組合せを用いた。Simics は機能シミュレーションを行うフルシステムシミュレータであり、また GEMS はメモリシステムの詳細なタイミングシミュレーションを担う。プロセッサ構成は 32 コアの SPARC V9 とし、OS は Solaris 10 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 microbench, SPLASH-2, [13] および STAMP[14] から計 10 個を使用した。なお、32 コア内の 1 コアは OS 用に割り当てる必要があるため、各ベンチマークプログラムは 31 スレッドで実行した。なお、本来 STAMP は 2 の冪乗数のスレッド数でしか動作しないベンチマークであるが、Gramoli ら [15] による、任意のスレッド数での実行を可能にする改変を施している。

5.2 評価結果

評価結果を図 7 および表 2 に示す。図 7 中の凡例はサイクル数の内訳を示しており、Non_trans はトランザクション外、Good_trans は結果的にコミットされたトランザクション内、Bad_trans は結果的にアボートされたトランザクション内、Aborting はアボート処理、Backoff はバックオフ処理、Stall はストール、Barrier はバリア同期、MagicWaiting は提案手法で追加した待機処理に要したサイクル数をそれぞれ示している。図中では、各ベンチマークプログラムの評価結果が 5 本のグラフで表されている。

これらのグラフは左から順に、それぞれ

(B) 既存の LogTM (ベースライン)

(R_1) RaR addr. のアドレス登録数を 1 とした提案モデル

(R_2) RaR addr. のアドレス登録数を 2 とした提案モデル

(R_4) RaR addr. のアドレス登録数を 4 とした提案モデル

(R_∞) アドレス登録数を限定しない参考モデル

の実行サイクル数の平均を表しており、既存の LogTM (B) の実行サイクル数を 1 として正規化している。ここで $(R_1) \sim (R_\infty)$ のアドレス登録数は、RaR addr. に登録可能な Read \rightarrow Write の順序でアクセスされるアドレスの数を示している。また、フルシステムシミュレータ上でマルチスレッドを用いた動作のシミュレーションを行うには、性能のばらつきを考慮する必要がある [16]。したがって、各評価対象につき試行を 10 回繰り返し、得られた結果から 95% の信頼区間を求めた。信頼区間はグラフ中にエラーバーで示す。なお、RaR addr. の参照・登録時に発生するオーバヘッドは非常に小さいため、ここには計上していない。詳細については 5.4 節で考察する。

評価の結果、多くのプログラムにおいて、ある共有変数に対し Write アクセスに先立って Read アクセスが行われるトランザクション処理が含まれており、Futile Stall を発生させる特徴を持っていることが確認できた。この Futile Stall を提案手法により解決することで、Btree を除く全てのプログラムで (B) 以上の性能が得られた。また、全体的に見られる傾向として、多くのプログラムで RaR addr. に登録するアドレスの数を多くした場合に、既存モデルに対する性能向上幅が大きくなっていることが分かる。しかし、アドレスの登録数を増やすことで得られる性能向上は目立ったものではなく、提案モデル (R_1) においても十分な性能向上が得られている。登録アドレス数を増加させるとハードウェアコストが増大することを考慮すると、 (R_1) が総合的に優れていると考えられる。この (R_1) において各ベンチマークプログラムを実行した場合、既存モデルに対して平均で 22.6%、最大で 66.9% の性能向上を得ることができた。次節では、各ベンチマーク別に検証を行う。

5.3 考察

GEMS microbench

まず GEMS microbench では、各提案モデルにおいて Deque, Prioqueue で実行サイクル数が減少しており、特に Backoff サイクル数の大幅な減少率が目立つ。これらのプ

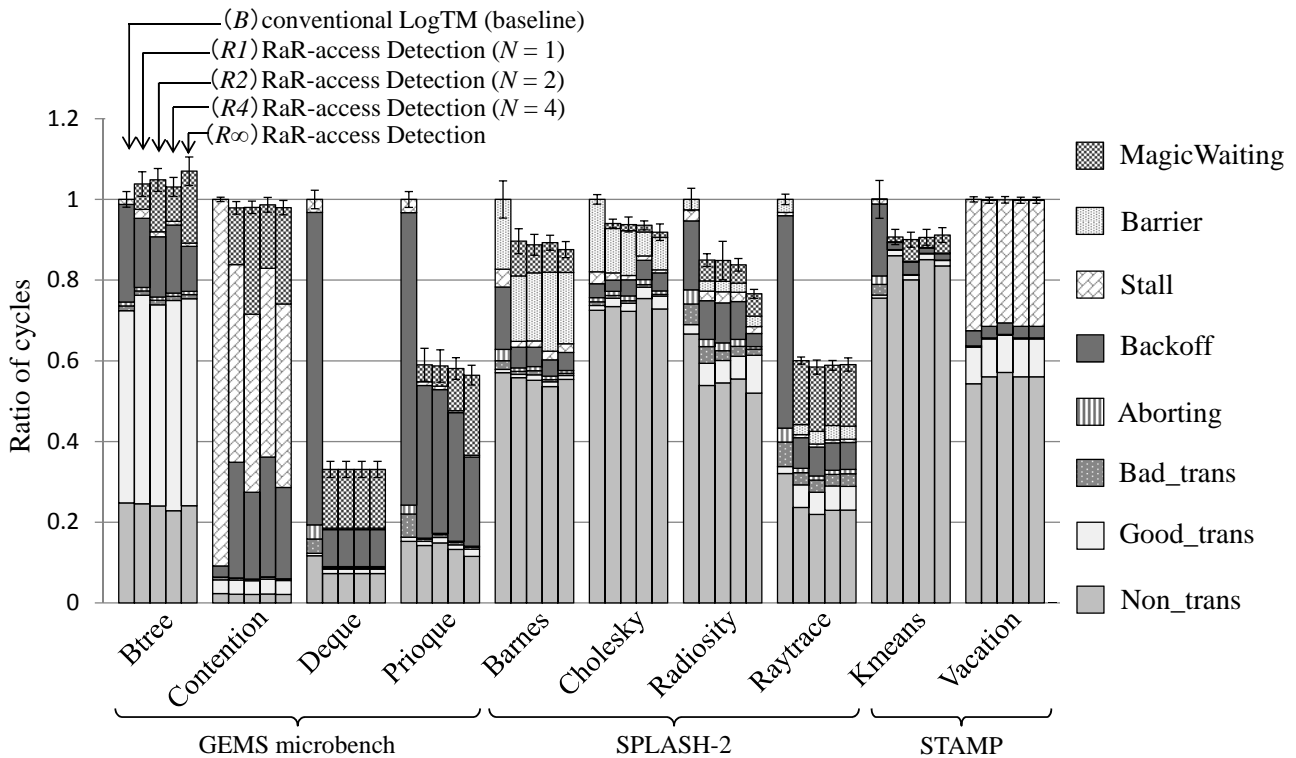


図 7 各プログラムにおけるサイクル数比

プログラムでは、ごく一部のアドレスのみが Read → Write の順序で頻繁にアクセスされたため、 (R_1) のようにアドレスの登録数が少なくても Futile Stall やそれに起因するアボートを十分抑制することができており、このことが Backoff サイクル数の大幅な削減につながった。

しかし、Btree を実行した場合にはどの提案モデルにおいても性能がわずかに低下した。この Btree には、2 種類のトランザクション (仮に $Tx.I$, $Tx.J$ とする) が存在し、 $Tx.I$ には Read → Write の順序でアクセスされるアドレスが含まれるが、 $Tx.J$ にはそのアドレスに対する Write アクセスは含まれておらず、Read アクセスのみが含まれている。そのため、複数の $Tx.I$ もしくは $Tx.I$ と $Tx.J$ が並列に実行される場合は本提案手法が効果的である。しかし、複数の $Tx.J$ のみが並列に実行される場合には Write アクセスが行われなため、Read アクセスを待機させることは適切ではない。これが原因で、提案モデルの性能がわずかに低下してしまったと考えられる。このような性能低下を防ぐために、並列実行すべきトランザクションの組み合わせを適切に判定することが考えられる。しかしこれを実現するためには、トランザクションの組合せ毎にアドレスの記憶領域を用意する必要があり、コストが膨大となるため、この性能低下に対処する必要性は低いと考えられる。

SPLASH-2

SPLASH-2 ベンチマークでは、各提案モデルにおいて Barnes, Cholesky, Raytrace の実行サイクル数が減少した。中でも Raytrace については Backoff サイクル数が大幅に減少している。このプログラム中にはあるアドレスに Read → Write の順序でアクセスする非常に短いトランザ

クションが 3 つ含まれており、既存モデルではこれらのトランザクションが原因で Futile Stall が頻発していた。したがって、これらのトランザクションを実行するスレッドに対して本手法を適用することで Futile Stall とそれに起因するアボートが抑制されたため、Backoff サイクルの大幅な削減につながった。また、Cholesky では Barrier サイクル数が減少した。これは、本手法により Futile Stall を抑制することで、各スレッドで発生するアボートの回数が減少し、実行を早く終了したスレッドが同期を行うために他のスレッドを待つ期間が短くなったためだと考えられる。

一方 Radiosity には、Read → Write の順序でアクセスされるアドレスが複数含まれており、これらのアドレスに対してアクセスが分散するため、各提案モデルにおいて、RaR addr. へのアドレス登録と登録されたアドレスの破棄が頻繁に行われていた。これにより、登録されたアドレスが早い段階で破棄されてしまう可能性が高くなり、正確に RaR を検出できなかった場合が多くあったと考えられる。したがって、Radiosity のようなプログラムに対する対処方法として、RaR addr. へのアドレス登録と破棄のアルゴリズムを改良することなどが挙げられる。

STAMP

STAMP ベンチマークでは、本手法によって Kmeans の実行サイクル数が減少した。このプログラム中には Read → Write の順序でアクセスされるアドレスが存在するが、Kmeans は他のベンチマークと比較して規模が小さいプログラムであり、本手法を適用して Futile Stall を抑制することによる性能向上の余地が少なかったと考えられる。

表 3 (R_1) における RaR addr. への参照・登録回数

GEMS	(R_1)	SPLASH-2	(R_1)	STAMP	(R_1)
Btree	876,235	Barnes	86,413	Kmeans	148,084
Contention	562,844	Cholesky	296,708	Vacation	684,826
Deque	7,152	Radiosity	115,865	-	-
Prioqueue	72,095	Raytrace	1,257,086	-	-

5.4 RaR addr. の参照コスト

4.2 節で述べた RaR addr. は RAM で構成されるハードウェアであるため、この RaR addr. への参照・登録の際にはオーバヘッドが発生する。そこで、そのオーバヘッドが性能にどの程度影響するのかについて検討する。まず、RaR addr. への参照・登録回数を C 、1 回の参照と登録で発生するオーバヘッドを T とすると、その総参照オーバヘッドは $C \times T$ という式で表すことができる。ここで、(R_1) における RaR addr. への参照・登録回数を調査した結果を表 3 に示す。4.2.3 項で示したとおり、RaR addr. は登録アドレス数が 1 の場合、256bytes という非常に小さい RAM で構成できる。このことから、各スレッドはこの RaR addr. のエントリに対して L1 キャッシュよりも高速にアクセスすることが可能であるため、RaR addr. のエントリには L1 キャッシュと同程度のオーバヘッド以下でアクセスできると考えられる。そこで本論文におけるシミュレータ構成から $T = 3\text{cycles}$ とし、最も参照・登録回数の多い Raytrace について、その総参照オーバヘッドを算出すると、 $1,257,086 \times 3 = \text{約 } 380 \text{ 万 cycles}$ となる。一方、Raytrace の総実行サイクル数は約 8.7 億 cycles であるため、このオーバヘッドが全体のサイクル数に占める割合は約 0.4% となり、性能に与える影響はごく僅かなものに抑えることができる。

6. おわりに

本稿では、Read \rightarrow Write の順序でアクセスされるアドレスを検出し、当該アドレスへの RaR アクセスに関わるトランザクションを逐次実行する手法を提案した。これにより、既存の HTM の性能を低下させる競合パターンである Futile Stall やこれに起因するアポートを抑制した。また、提案手法を実現するために追加したハードウェアのコストはいずれも少ないことが分かった。提案手法の有効性を確認するために GEMS microbench, SPLASH-2 および STAMP ベンチマークプログラムを用いて評価した結果、既存の HTM と比較して最大 66.9%、平均 22.6% の実行サイクル数が削減されることを確認した。しかし提案モデルでは、トランザクションを並列実行すべき状況でも、それらを逐次的に実行してしまう場合があった。したがって、今後逐次実行すべきトランザクションをより適切に選択する必要がある。また、提案モデルでは実行順序制御時に遊休状態となるスレッドが存在するため、そのようなスレッドに対して別の処理をさせることで並列性の向上を図る方法について検討することも今後の課題である。

参考文献

- [1] Herlihy, M. et al.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. 20th Int'l Symp. on Computer Architecture (ISCA'93)*, pp. 289–300 (1993).
- [2] J.Moravan, M. et al.: Supporting Nested Transactional Memory in LogTM, *Proc. 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 1–12 (2006).
- [3] Moss, E. and Hosking, T.: Nested Transactional Memory: Model and Preliminary Architecture Sketches., *In OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages*, pp. 186–201 (2005).
- [4] M, L., G, M. and A, G.: A Dynamically Adaptable Hardware Transactional Memory, *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM*, pp. 27–38 (2010).
- [5] Shriraman, A., Dwarkadas, S. and Scott, M. L.: Flexible Decoupled Transactional Memory Support, *ISCA '08 Proceedings of the 35rd annual international symposium on Computer Architecture*, pp. 139–150 (2008).
- [6] Yoo, R. M. and Lee, H.-H. S.: Adaptive Transaction Scheduling for Transactional Memory Systems, *Proc. 20th Annual Symp. on Parallelism in Algorithms and Architectures (SPAA'08)*, pp. 169–178 (2008).
- [7] Blake, G., Dreslinski, R. G. and Mudge, T.: Bloom Filter Guided Transaction Scheduling, *Proc. 17th International Conference on High-Performance Computer Architecture (HPCA-17 2011)*, pp. 75–86 (2011).
- [8] Akpinar, E., Tomić, S., Cristal, A., Unsal, O. and Valero, M.: A Comprehensive Study of Conflict Resolution Policies in Hardware Transactional Memory, *Proc. 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'11)* (2011).
- [9] Gaona, E., Titos, R., Acacio, M. E. and Fernández, J.: Dynamic Serialization Improving Energy Consumption in Eager-Eager Hardware Transactional Memory Systems, *Proc. Parallel, Distributed and Network-Based Processing 2012 20th Euromicro International Conference (PDP'12)*, pp. 221–228 (2012).
- [10] Moore, K. E. et al.: LogTM: Log-based Transactional Memory, *Proc. 12th Int'l Symp. on High-Performance Computer Architecture*, pp. 254–265 (2006).
- [11] Magnusson, P. S. et al.: Simics: A Full System Simulation Platform, *Computer*, Vol. 35, No. 2, pp. 50–58 (2002).
- [12] Martin, M. M. K. et al.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset, *ACM SIGARCH Computer Architecture News*, Vol. 33, No. 4, pp. 92–99 (2005).
- [13] Woo, S. C. et al.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd Int'l. Symp. on Computer Architecture (ISCA'95)*, pp. 24–36 (1995).
- [14] Minh, C. C., Chung, J., Kozyrakis, C. and Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing, *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08)* (2008).
- [15] Gramoli, V. et al.: Transactions, <http://lpdservr.epfl.ch/transactions/wiki/doku.php?id=stamp> (2011).
- [16] Alameldeen, A. R. and Wood, D. A.: Variability in Architectural Simulations of Multi-Threaded Workloads, *Proc. 9th Int'l Symp. on High-Performance Computer Architecture (HPCA'03)*, pp. 7–18 (2003).