情報処理学会論文誌　コンピューティングシステム　Vol.6　No.4　58–71 (Oct. 2013)

Read-after-Readアクセスの制御による ハードウェアトランザクショナルメモリの高速化

橋本 高志良[1]　堀場 匠一朗[1]　江藤 正通[1,†1]　津邑 公暁[1,a)]　松尾 啓志[1]

受付日 2013年4月9日, 採録日 2013年7月4日

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，その後に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あ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16スレッド並列実行時において最大53.6%，平均15.6%の高速化が得られることを確認した．

キーワード：ハードウェア・トランザクショナル・メモリ，スレッドスケジューリング，Read-after-Readアクセス，競合解決

A Speed-up Technique for Hardware Transactional Memory by Controlling Read-after-Read Accesses

Koshiro Hashimoto[1]　Shoichiro Horiba[1]　Masamichi Eto[1,†1]　Tomoaki Tsumura[1,a)]
Hiroshi Matsuo[1]

Received: April 9, 2013, Accepted: July 4, 2013

Abstract: Lock-based thread synchronization techniques are commonly used in parallel programming on multi-core processors. However, lock can cause deadlocks and poor scalabilities. Hence, Transactional Memory (TM) has been proposed and studied for lock-free synchronization. On TM, transactions are executed speculatively unless a memory access conflict is caused, hence the performance of TM is generally better than that of lock. However, if speculative execution is continued when a Read-after-Read (RaR) access occurs, following stalls can be wasted. In this paper, we propose an effective thread scheduling by controlling some RaR accesses. On our proposal, when a RaR access to some particular address is detected between some transactions, the execution of those transactions is serialized. The result of the experiment shows that proposed method improves the performance 53.6% in maximum and 15.6% in average.

Keywords: Hardware Transactional Memory, thread scheduling, Read-after-Read access, contention management

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ともなう並

[1] 名古屋工業大学
Nagoya Institute of Technology, Nagoya, Aichi 466–8555, Japan
[†1] 現在，東海旅客鉄道株式会社
Presently with Central Japan Railway Company
[a)] tsumura@computer.org

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やすいものではない．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（**version management**; **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つねに検査する必要がある（**conflict detection**; **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 アクセス済みである変数に対し，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うる 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あ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 つの手法があげられる．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ている．一方で，Blake ら [7] は複数のトランザクショ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ている．また，Akpinar ら [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ている．

もう一方の方向からの改良として，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ている．

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のみに基づいてスレッドの振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 の性能を低下させう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．

一方で，共有メモリマルチプロセッサにおける効率的なロック獲得制御手法として，QOLB（Queue On Lock Bit）[10] がある．これは，ロック獲得要求をキューで管理することで，プロセッサ間でロック権限の効率的な受け渡しを実現するものである．本稿では，この QOLB と同様な発想により，HTM において共有リソースへのアクセス要求順序をキューで管理することで，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. Read-after-Read アクセスの制御

本章では，既存の HTM における問題点と，それを解決する提案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

ロックを用いる場合，クリティカルセクションとして定義された範囲は，完全に排他的かつ逐次的に実行される．一方 TM では，トランザクションとして定義された処理は，競合が検出されない限り投機的に並列実行されるため，TM はロックよりも高い性能を実現できるとされている．

ここで，TM において競合として検出されるアクセスは，Write アクセスが関与する Read-after-Write（RaW），Write-after-Read（WaR），および Write-after-Write（WaW）の 3 パターンであり，Read-after-Read（RaR）は競合としては検出されない．しかし，この RaR アクセスを許可しトランザクションの投機的実行を継続させることが，結果的に TM の処理に大きな無駄を発生させる場合がある．

一般に，共有変数への Read アクセスは，その後に Write アクセスをともな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

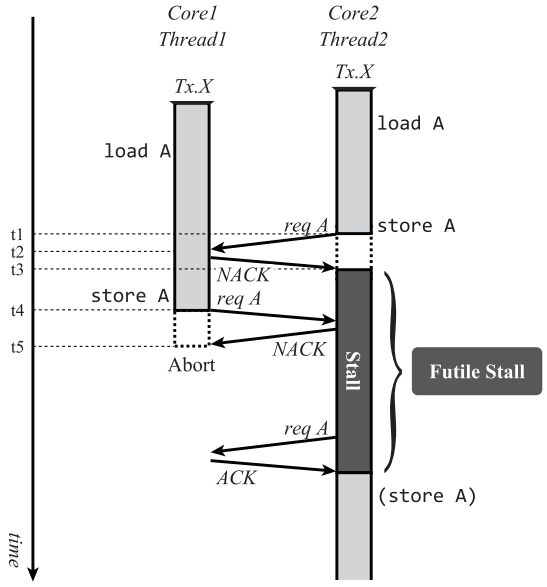

図 1　Read-after-Read アクセスに起因する Futile Stall
Fig. 1　Futile stall derived from a Read-after-Read access.

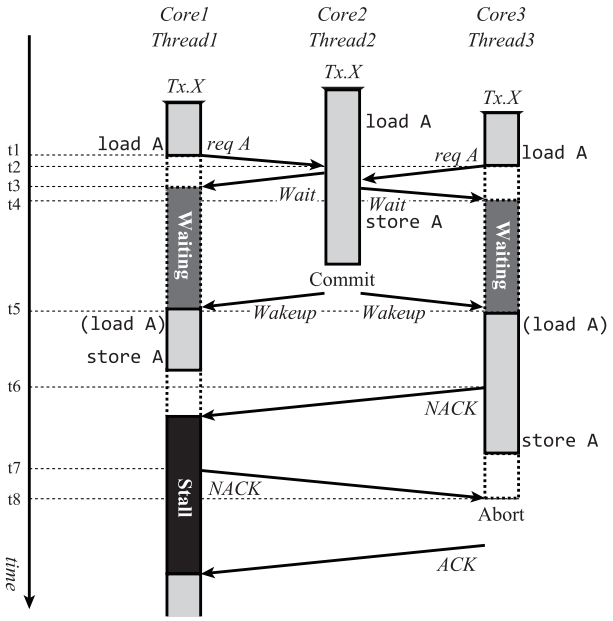

図 2　RaR アクセスの制御による Futile Stall の抑制
Fig. 2　Avoiding futile stall by disapproving Read-after-Read access.

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うる．

図 1 は，上記のような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` を実行しようとした際（時刻 t1），競合が検出される（t2）．ここで LogTM [11] に代表される，eager conflict detection を採用する HTM では一般に，競合を起こした *Thread2* が，*NACK* の受信にともない自身の *Tx.X* をストールする（t3）．その後，*Thread1* が `store A` を実行しようとする際（t4），*Thread2* はすでに当該アドレスにアクセス済みであるため競合を検出し，*Thread1* へ *NACK* を返信する．このとき，*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（t5）．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* の実行はいっさい進行しておらず，*Thread2* のストールは完全に無駄であったことになる．このように，結果的にアボートしてしまうようなトランザクションとの競合により発生する無駄なストールは **Futile Stall** [12] と呼ばれ，HTM のスループットを低下させる大きな要因となる．

3.2　Read-after-Read アクセス制御手法

本節では，Read-after-Read アクセスの制御により Futile Stall を解決する手法を提案する．

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みと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，すでに Read アクセス済みであった他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，待機させたアクセスを順次許可する手法を提案する．

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*〜*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とする（時刻 t1，t2）．これがキャッシュミスとなった場合，Read リクエストが送信されるが，この際 *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 は解決できるが，*Thread2* は実行トランザクションをコミットした際，*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，t5）．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，これらをいっせい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（t6，t7）．なお，簡略化のため図 2 においては，時刻 t5 以降のアドレス `A` に対するリクエストは省略している．その後，発生した競合により *Thread3* が *Tx.X* をアボートするため（t8），*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あわせて提案する．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，記憶した順序に従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．

なお，以上のように動作させると，再開順序制御に関与しているスレッド（*Thread1*～*Thread3*）が，再開順序制御に関与していない他のスレッドと競合する場合がある．たとえば，*Thread1* が，*Thread2*～*Thread3* 以外のスレッドと競合して *Tx.X* をストールさせたとすると，このストールは *Thread2*～*Thread3* の実行時間にも影響を与えてしまう．そこで本提案手法では，再開順序制御に関与しているスレッドが，これに関与していないスレッドと競合した場合，後者のスレッドがトランザクションをアボートし，前者のスレッドがトランザクションを優先的に実行することとする．

なお，本稿で提案する手法は，競合発生時にストールを用いるような HTM 実装において，無駄なストールの発生を抑制するものである．一方**図 3** に示すように，ストールを用いず，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ことでも競合は回避できる．しかしこの場合，アボートの多発により，アプリケーションによっては大きな性能低下を引き起こしうると考えられる．本稿では提案手法の有効性を確認するため，このストールを用いない実装との比較も 5 章で行う．

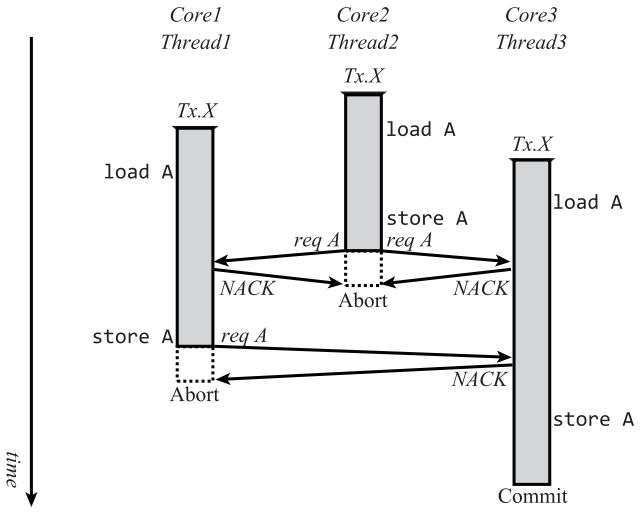

図 3　Write アクセスを試みたトランザクションをアボートさせる動作例

Fig. 3　Aborting writer transactions.

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 3 つのユニットを各コアに追加する．

Register for target addresses（Tgt-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，Tgt-addr. に記憶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，本レジスタに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，Tgt-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，Tgt-addr. を参照して RaR アクセスを検出すべき

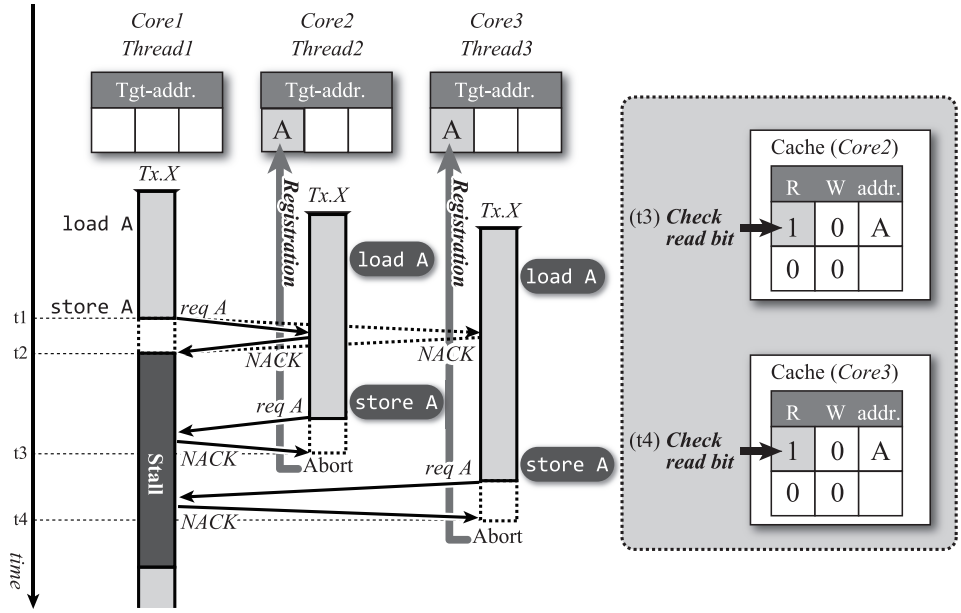

図 4　RaR アクセスを検出すべきアドレスの検知と Tgt-addr. への記憶
Fig. 4　Detection and registration of target addresses.

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．

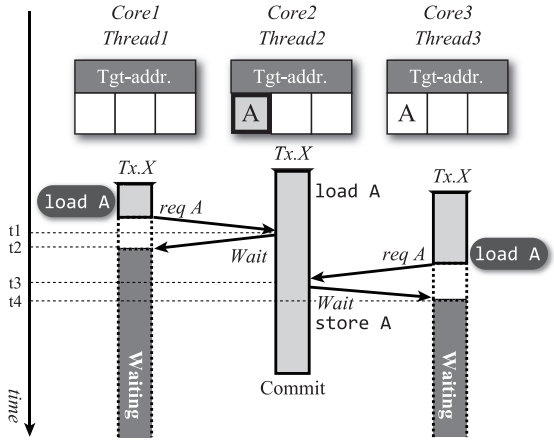

図 5　Tgt-addr. を利用した RaR アクセスの検出
Fig. 5　RaR access detection using Tgt-addr. register.

4.2　Read-after-Read アクセス検出の実現

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　Tgt-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 4** を例に，追加した Tgt-addr. へのアドレス登録の動作を説明する．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 t1），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（t2）．続いて，*Thread2* と *Thread3* がそれぞれ `store A` を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．このとき，*Thread2* と *Thread3* はアクセスしようとしていたアドレス `A` に対応する自身の 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ごと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 Tgt-addr. に登録する．

4.2.2　Tgt-addr. の利用

4.2.1 項で述べた方法で Tgt-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 5** 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 Tgt-addr. を参照することとする．

図 5 の例では，すでに *Thread2* の Tgt-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．このとき，*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．この *req A* を受信した *Thread2* は，自身の Tgt-addr. を参照し，アドレス A が登録済みかどうかを確認する（時刻 t1）．*Thread2* の Tgt-addr. には当該アドレス A がすで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　Tgt-addr. のハードウェアコスト

ここで，Tgt-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，Tgt-addr. には Read→Write の順序でアクセスされたアドレスが記憶される．しかし，プログラム中において Read→Wrire の順序でアクセスされるアドレスをすべ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，Tgt-addr. に記憶できるアドレス数を最大 N としてコストを抑える．仮に記憶アドレス数 N を 1，2，または 4 と設定した場合，それぞれコアあたり 64 bit，128 bit，256 bit 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 256 byte，512 byte，1 Kbyte と少量で実現できる．なお，記憶可能なアドレス数を複数とした場合，記憶アドレスの管理方法はいくつかの選択肢をとりうるが，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 Tgt-addr. の記憶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，記憶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 6** に示す．これは，図 5 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登録する．一方，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であると判断し，自身の R-res. に *Thread2* を実行するコア番号を登録する．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登録する．そして，*Thread3* は R-res. に *Thread2* を実行するコア番号を登録する．

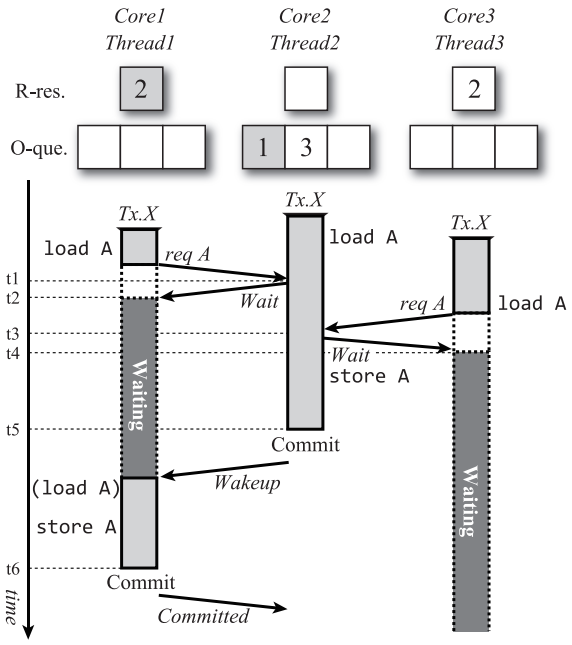

図 6　O-que. と R-res. の利用
Fig. 6　Utilizing O-que. and R-res.

次に O-que. と R-res. に登録されたコア番号を利用して，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. に登録されている番号をチェックする．このとき，*Thread2* の O-que. にはコア番号 1，3 が順に登録されており，*Thread2* は 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，*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. に登録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 7** 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*

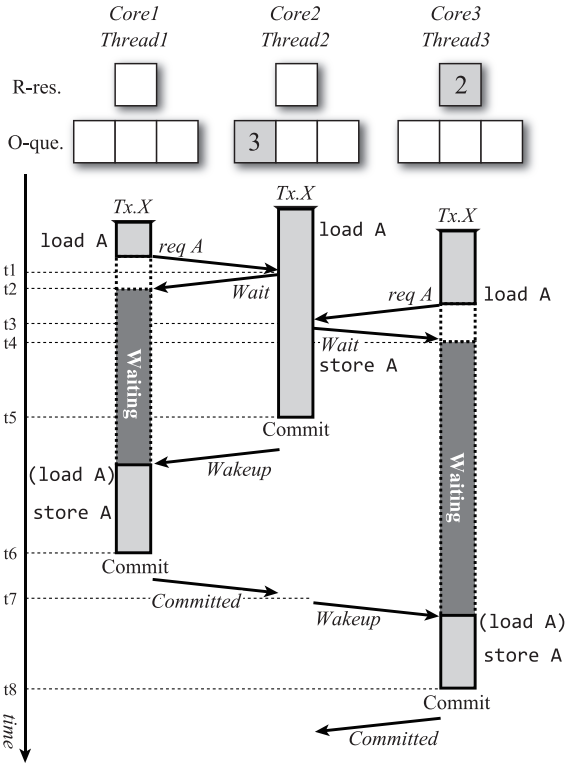

図 7　再開順序制御によるトランザクションの逐次実行
Fig. 7　Serialization of transactions by controlling the order of precedence.

通知を受信した *Thread2* は，再度自身の O-que. をチェックする．このとき，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すべ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. のハードウェアコストについて検討する．これらのエントリにはコア番号が記憶されるため，32 コア構成のプロセッサの場合 1 エントリあたり 4 bit 必要となる．また，O-que. には，最大で自コアを除くすべてのコア番号を記憶する必要があるため，4 bit × 31 の記憶容量が必要となる．以上のことから，必要となる総記憶容量は，4 bit × 32 × 32 = 512 bytes と少量である．

5. 性能評価

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

これまで述べた提案手法を，HTM の研究で広く用いられている LogTM [11] 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 Simics [13] 3.0.31 と GEMS [14] 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．

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	1 cycle
D2 cache	8 MBytes
ways	8 ways
latency	20 cycles
Memory	8 GBytes
latency	450 cycles
Interconnect network latency	14 cycles

評価対象のプログラムとしては GEMS 付属 microbench，SPLASH-2 [15]，および STAMP [16] から計 10 個を使用し，各ベンチマークプログラムを 16 および 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のみ動作するベンチマークであるが，Gramoli らによる，任意のスレッド数での実行を可能にする改変 [17] を施すことで，31 スレッドでも実行した．

5.2 評価結果

31 スレッド実行による評価結果を**図 8** および**表 2** に，16 スレッド実行による評価結果を**図 9** および**表 3** に示す．図 8 および図 9 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の実行サイクル数，Good_trans はコミットされたトランザクションの実行サイクル数，Bad_trans はアボートされたトランザクションの実行サイクル数，Aborting はアボート処理に要したサイクル数，Backoff はバックオフ処理に要したサイクル数，Stall はストールに要したサイクル数，Barrier はバリア同期に要したサイクル数，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．

また図中では，各ベンチマークプログラムの評価結果が 5 本のバーで表されているが，これらのバーは左から順に，

(B)　既存の LogTM（ベースライン）
(R_1)　Tgt-addr. のアドレス記憶数を 1 とした提案モデル
(R_2)　Tgt-addr. のアドレス記憶数を 2 とした提案モデル
(R_4)　Tgt-addr. のアドレス記憶数を 4 とした提案モデル
(R_∞)　アドレス記憶数に上限を与えない参考モデル

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)〜(R_∞) のアドレス記憶数とは，Tgt-addr. に記憶可能な Read→Write の順序でアクセスされるアドレスの数を示している．なお，フルシステムシミュレータ上でマル

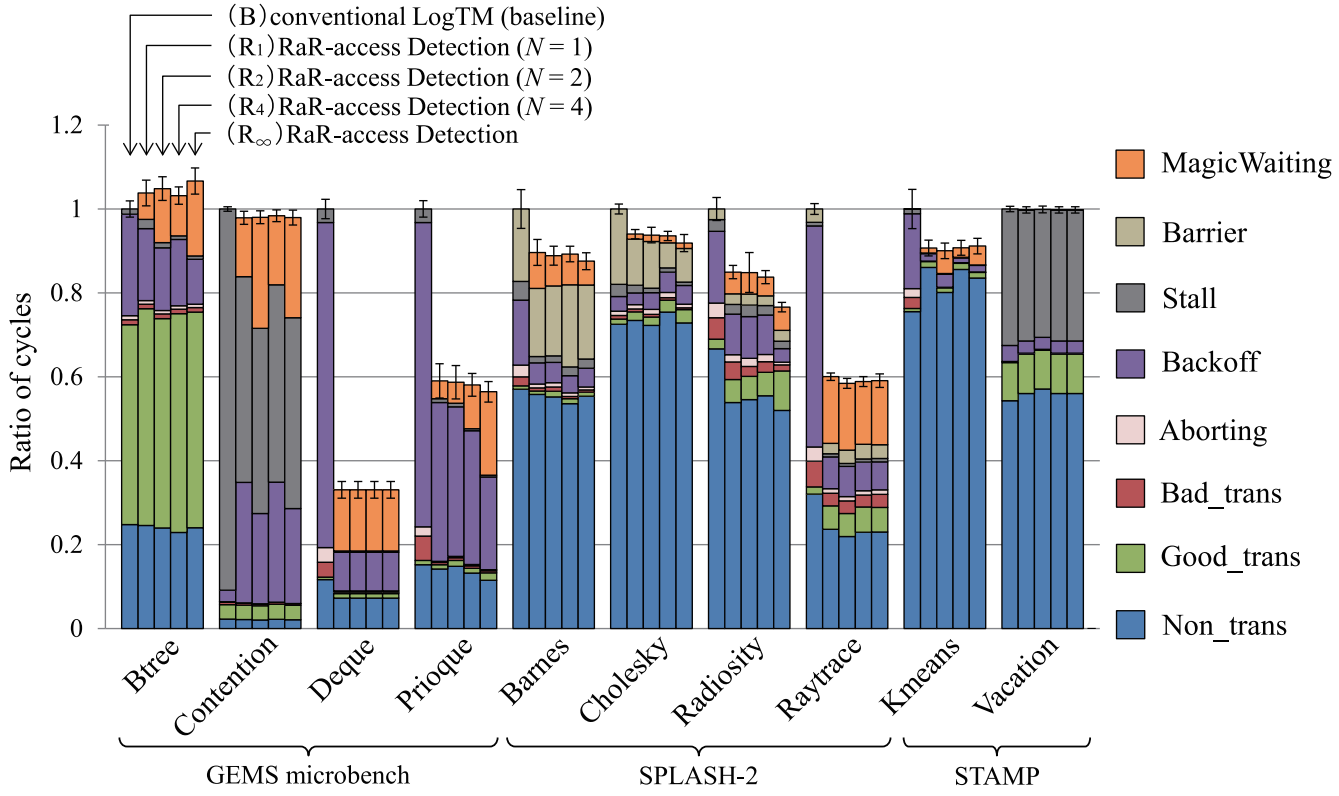

図 8　各プログラムにおけるサイクル数比（31 スレッド実行）

Fig. 8　Execution cycles ratio (31 threads).

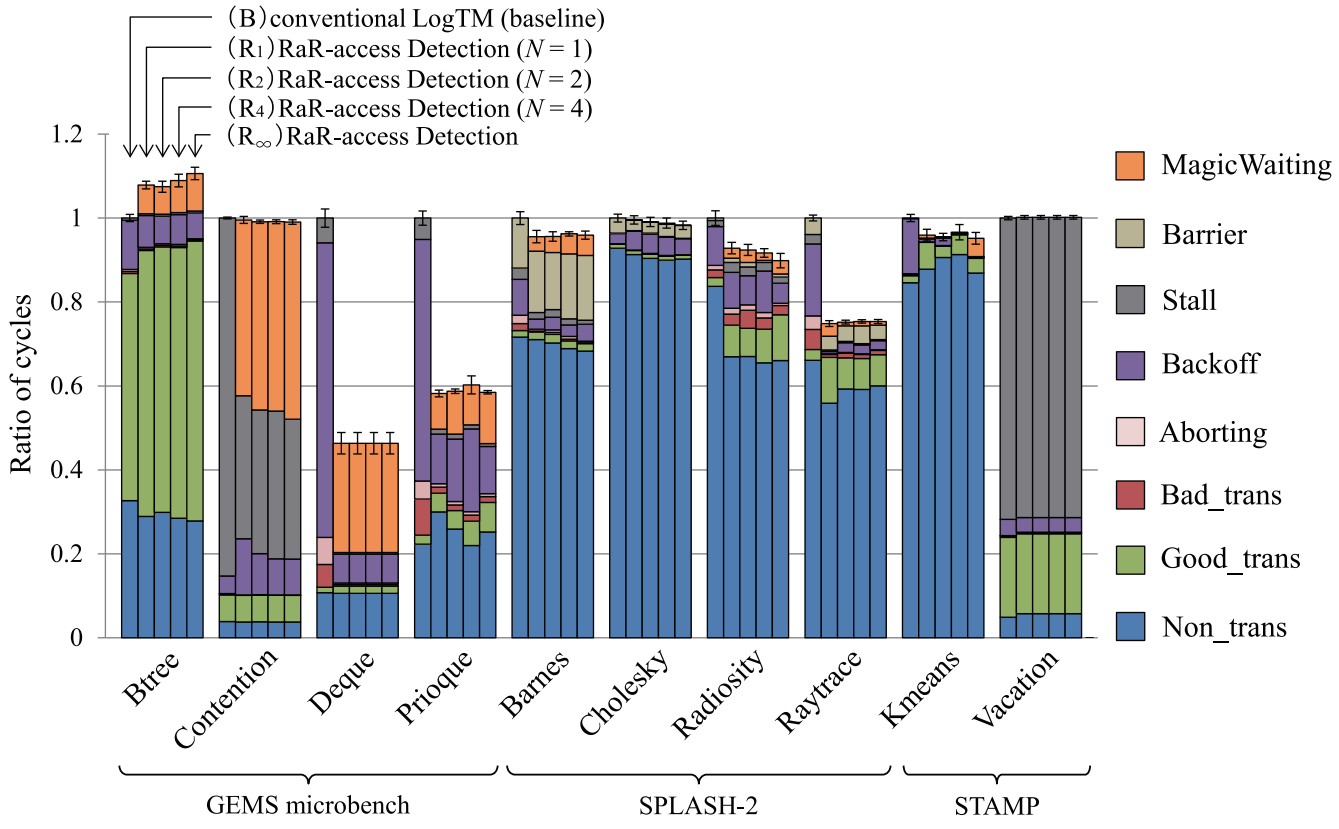

図 9　各プログラムにおけるサイクル数比（16 スレッド実行）

Fig. 9　Execution cycles ratio (16 threads).

表 2　各ベンチマークにおけるサイクル削減率（31 スレッド実行）
Table 2　Reduced cycles ratio (31 threads).

		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%

表 3　各ベンチマークにおけるサイクル削減率（16 スレッド実行）
Table 3　Reduced cycles ratio (16 threads).

		GEMS	SPLASH-2	STAMP	All
(R_1)	平均	22.7%	9.9%	2.0%	15.5%
	最大	53.6%	25.1%	4.0%	53.6%
(R_2)	平均	22.7%	10.0%	2.2%	15.6%
	最大	53.6%	24.8%	4.5%	53.6%
(R_4)	平均	22.0%	10.0%	1.7%	15.3%
	最大	53.6%	24.6%	3.4%	53.6%
(R_∞)	平均	22.2%	10.7%	2.3%	15.8%
	最大	53.6%	24.6%	4.8%	53.6%

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 [18]．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，提案手法実現のために追加した 3 つのユニットへのアクセス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．このオーバヘッドについては，5.5 節で別途考察する．

評価結果から，16 スレッドおよび 31 スレッド実行のいずれにおいても，大幅な性能向上が得られていることが分かる．このことから，多くのプログラム中には，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 を発生させうるという特徴があ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 を除くすべてのプログラムで (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，Tgt-addr. に記憶できるアドレス数を増加させるにつれ，多くのプログラムで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，アドレス記憶数を増加させる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．また，アドレス記憶数を増加させると，それにともなってハードウェアコストも増大することを考慮すると，(R_1) が性能およびコストの観点から見て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 31 スレッドで実行した場合，既存モデルに対して平均 22.6%，最大 66.9%，また 16 スレッドで実行した場合，既存モデルに対して平均 15.6%，最大 53.6%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詳細な検証を行う．

5.3　考察

GEMS microbench

まず GEMS microbench では，各提案モデルにおいて Deque，Prioque で実行サイクル数が大きく減少しており，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記憶数が少なくても，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 の大幅な削減につながったと考えられる．

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，ツリー構造に対する挿入および検索操作のための 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* どうし，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 アクセスが発生しないため，Read アクセスを待機させることは適切ではない．Btree ではそのような無駄な待機時間が多く発生していたため，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 Btree のように，あるデータ構造に対する挿入/検索トランザクションが並行動作するような場合には，同様の性能低下が引き起こされる可能性が高く，データ構造が大きくトランザクション処理時間が長い場合に，特にその性能低下が顕著になってしまうと考えられる．よって今後，Read アクセスのみを行うトランザクションを例外的に扱うための仕組みを検討する必要があると考えられる．

なお，Contention，Deque，Prioque において，16 スレッド実行時よりも 31 スレッド実行時の方が MagicWaiting の割合が減少しているが，この理由について補足する．まず Contention に関しては，31 スレッド実行において Backoff が大きく増大していることから，並列度の増加にともなってアボートの繰返しが多発していることが分かる．この Backoff の増大により同一トランザクションどうしが並列実行される場面が少なくなり，**表 4** に示すように，Read→Write の順にアクセスされるアドレスに対する RaR アクセス自体が減少したことが原因である．一方 Deque および Prioque については，16 スレッド実行よりも 31 スレッド実行の方が総実行サイクル数が多くなって

表 4　Contention における平均 RaR アクセス検出回数
Table 4　Detected RaR access average count in Contention.

	16 threads	31 threads
RaR アクセス検出回数	94.0	66.2

表 5　Deque および Prioque における Magic Waiting のサイクル数
Table 5　Magic Waiting cycles of Deque and Prioque.

(R_1)	16 threads	31 threads
Deque	1,211,862	1,548,405
Prioque	295,077	395,844

```
1 BEGIN_TRANSACTION( 16 );
2   ray->id = gm->rid++;
3 COMMIT_TRANSACTION( 16 );
```

図 10　Raytrace プログラム内のトランザクション
Fig. 10　A transaction in Raytrace benchmark program.

しまうプログラムであり，MagicWaiting の割合は減少しているものの，**表 5** に示すように MagicWaiting のサイクル数自体は増加していた．

SPLASH-2

SPLASH-2 ベンチマークでは，各提案モデルにおいてすべてのプログラムの実行サイクル数が減少した．これらの中でも Raytrace については，Backoff サイクル数が大幅に減少している．Raytrace のプログラム中に存在する，同じアドレスに Read→Write の順序で頻繁にアクセスするトランザクションの 1 つを**図 10** に示す．このトランザクション内では，`gm->rid` 変数のインクリメントが行われており，これが同一変数への Read→Write 順のアクセスにあたる．ユニークな ID を生成するために，この変数のインクリメントは排他的に行われる必要がある．複数スレッドがこのトランザクションを並行実行した場合，既存モデル (B) では図 1 で示したように Futile Stall が発生し，なおかつ一方のトランザクションがアボートされることから Backoff サイクルも増大する．Backoff は再競合を回避するために再実行までの待ち時間を設ける目的があるが，競合が多発すると指数関数的に増大するため，必要以上に大きな値となってしまう場合がある．一方で提案手法を用いた場合，Futile Stall を抑制できるだけでなく，*Wakeup* メッセージにより最適な待ち時間でトランザクションを再開させることができ，待ち時間の無駄も解消されることが性能向上につながっている．

また，Cholesky では Barrier サイクル数が有意に減少している．これは，本提案手法により 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，実行を早く終え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，Tgt-addr. へのアドレス登録と，記憶されたアドレスの破棄とが頻繁に行われていた．これにより，記憶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 アクセス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，Tgt-addr. へのアドレスの登録および破棄のアルゴリズムを改良することなどがあげられる．

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るため，本手法の目的である，Futile Stall の抑制による性能向上の余地が少なかったと考えられる．

5.4　ストールを用いない競合解決手法との比較

3.2.2 項末でも述べたように，本提案手法は，競合発生時にストールを用いる HTM 実装において，Futile Stall の発生を抑制するものである．一方で，ストールを用いず，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実装の場合，このような Futile Stall はそもそも発生しえない．本節では，上記のようなアボートのみを用いることで競合解決を図る手法と比較することで，本提案手法の有効性を確認する．

ここでは，競合時にアボートを用いる 2 つの比較対象モデル (A_W)，(A_R) を定義する．ただし，本提案手法が適用される場面における振舞いの違いが性能に与える影響を確認するため，これらのモデルでは，WaR 競合発生時に限り，それぞれ writer トランザクションまたは reader トランザクションを即座にアボートさせることとし，RaW/WaW 競合時には通常どおりストールを用いることとする．このモデルの 31 スレッドおよび 16 スレッドによる実行サイクル数を前項の (B) および (R_1) と比較した結果を，**図 11** および**図 12** にそれぞれ示す．前項のグラフと同様に，(B) の実行サイクル数を 1 として正規化しており，内訳の凡例も同じである．

評価の結果，ほぼすべてのベンチマークプログラムにおいて，提案モデル (R_1) が (A_W) および (A_R) の性能を上回っていることが確認できた．プログラム別に詳細に見ると，まず Deque や Prioque において (A_W) および (A_R) は，(R_1) には及ばないものの，ある程度の性能向上が得られていることが分かる．これらのプログラムを (A_W) および (A_R) により実行した場合，writer もしくは reader トランザクションがアボートを繰り返す．しかしこれらのプログラムに含まれるトランザクションは非常に短いため，再実

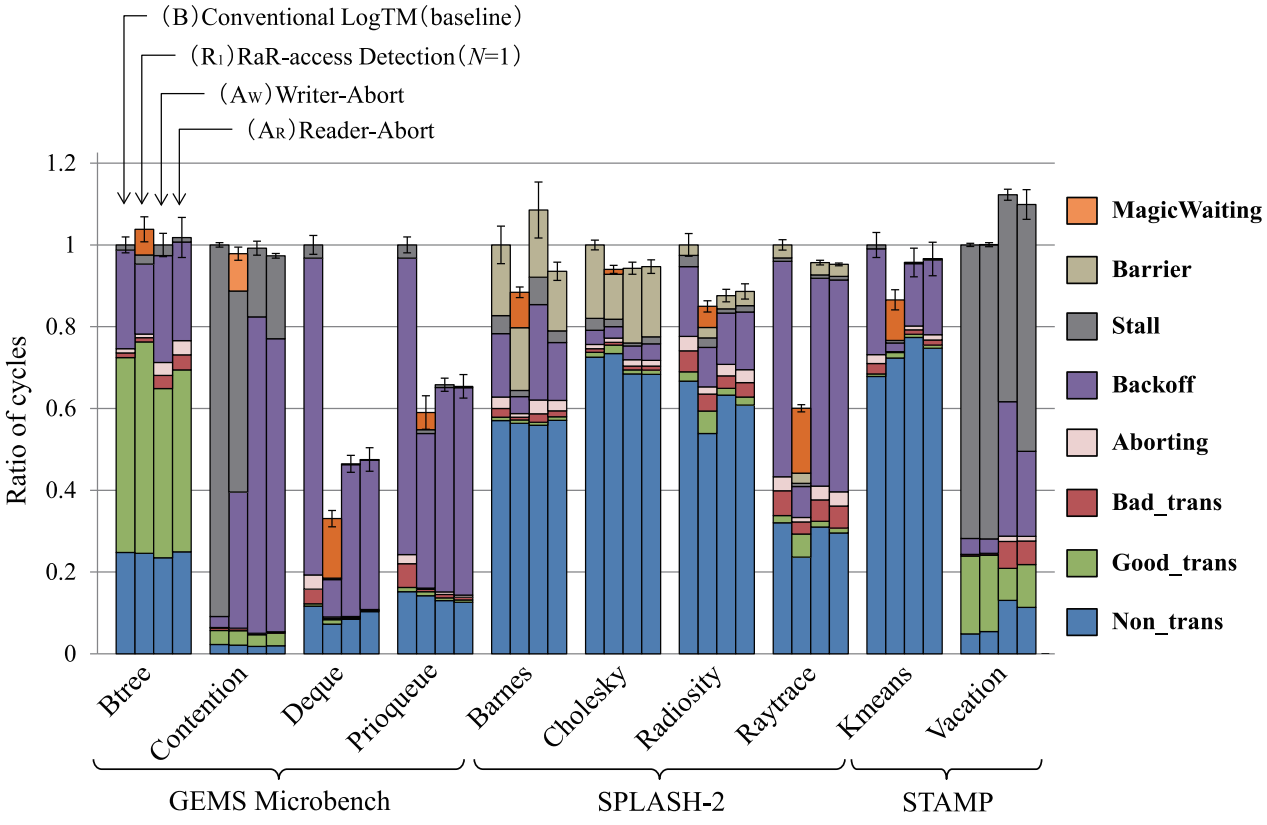

図 11　各プログラムにおけるサイクル数比（31 スレッド実行）

Fig. 11　Execution cycles ratio (31 threads).

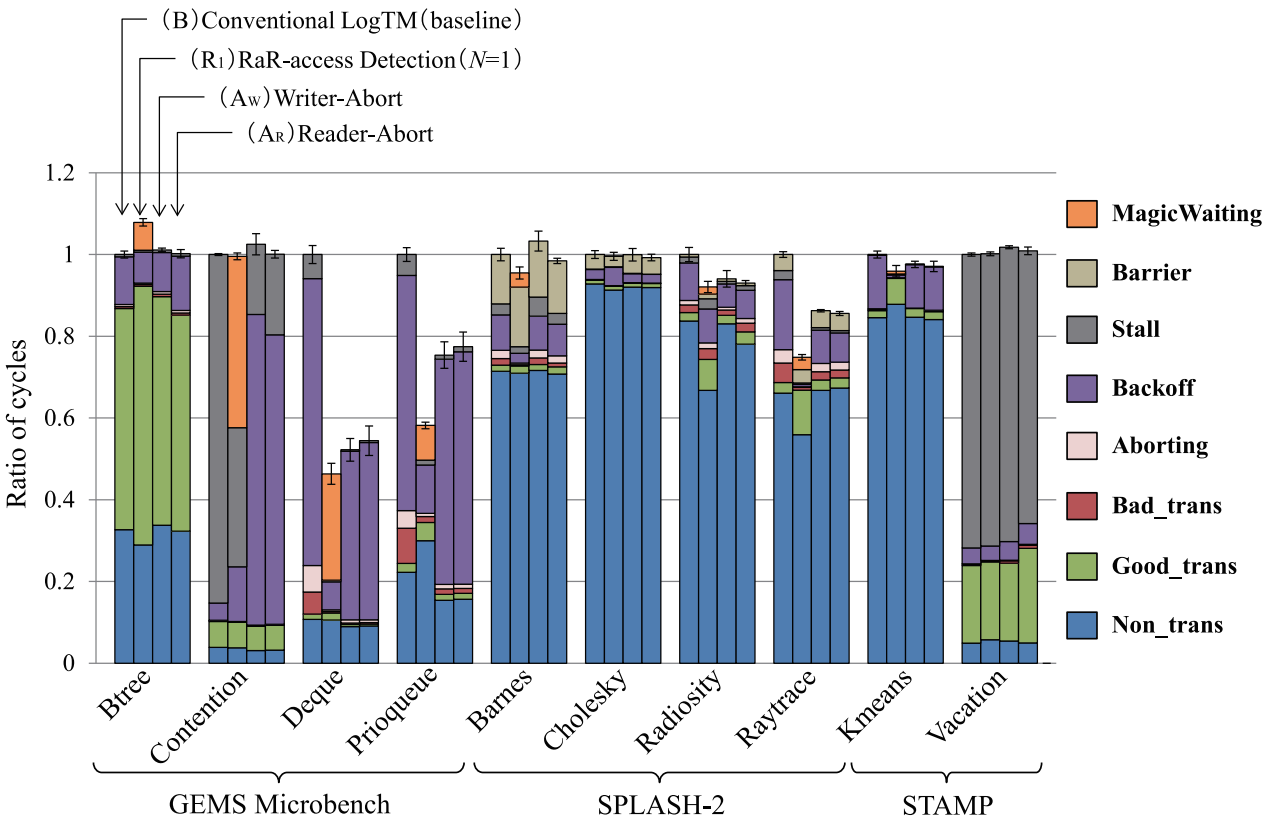

図 12　各プログラムにおけるサイクル数比（16 スレッド実行）

Fig. 12　Execution cycles ratio (16 threads).

```
1  BEGIN_TRANSACTION( 2 );
2   if (*qptr == NULL) {
3     le = InitLeaf((cellptr) mynode, ProcessId);
4     Parent(p) = (nodeptr) le;
5     Level(p) = l;
6     ChildNum(p) = le->num_bodies;
7     ChildNum(le) = kidIndex;
8     Bodyp(le)[le->num_bodies++] = p;
9     *qptr = (nodeptr) le;
10    flag = FALSE;
11  }
12 COMMIT_TRANSACTION( 2 );
```

図 13　Barnes プログラム内のトランザクション
Fig. 13　A transaction in Barnes benchmark program.

行による無駄な実行サイクルの増加はあまり大きくならない．また，アボートの繰返しによりバックオフ期間が増大することで，提案手法における実行待機と同様の効果が得られており，ある程度性能が向上している．しかし，バックオフよりも提案手法の実行待機の方が適切な待機時間が設定されるため，提案手法の方が有意に性能が勝っている．

一方で，Barnes，Raytrace，Vacation においては，(A_W) および (A_R) は，(R_1) と比較して性能が大きく劣っており，(B) より性能悪化してしまう場合もあることが分かる．これらのプログラム中には，上述した Deque や Prioque とは異なり，ある程度長く，かつ競合が発生しやすいトランザクションも含まれている．そのため，バックオフによる効果と提案手法における実行待機による効果の差に開きが生まれたと考えられる．また一部のプログラムでは Bad_trans も増加していることから，アボート後の再実行にともなう無駄な実行サイクルも性能に悪影響を及ぼしていると考えられる．

なお，(A_W) と (A_R) の性能を比較すると，多くのプログラムにおいてほぼ同等の性能となっているが，Barnes の 31 スレッド実行など，有意に (A_R) の性能が勝る場合があることが分かる．Barnes 内で多く競合を引き起こすトランザクションの 1 つを，**図 13** に示す．このトランザクションでは，変数 `*qptr` に対して Read→Write 順のアクセスが行われているが（2 行目および 9 行目），その間にはある程度の処理が含まれている．このような場合において，(A_W) のように writer 側（*ThreadA* とする）のトランザクションをアボートさせ reader 側（*ThreadB* とする）を継続させると，トランザクションを再実行した *ThreadA* が，*ThreadB* のトランザクションがコミットに至る前に再度共有変数に read アクセスしてしまうことで立場が逆転し，*ThreadB* の write リクエスト時に今度は *ThreadB* のトランザクションがアボートされてしまう．これを繰り返すことで，いずれのトランザクションもコミットに至らないままお互いを必要以上にアボートさせあう状況が発生し，それにともないバックオフも大きく増大することで，(A_R) に比べて大きな性能低下を引き起こしうる．一方で，図 10 に示した Raytrace のトランザクションのように，Read アクセスと Write アクセスの間隔が短くトランザクションも短い場合，reader/writer いずれのトランザクションをアボートした場合においても，十分短いバックオフ期間中に他方のトランザクションがコミットに至る可能性が高いと考えられる．このため，(R_1) よりはアボートが発生するものの，(A_W) においても処理がいっさい進行しないままアボートが繰り返されるような状況には陥らないため，(A_R) との性能に差が生じていないと考えられる．

表 6　ストールを用いないモデルとのサイクル削減率の比較
Table 6　Reduced cycles ratio of aborting models.

		31 threads	16 threads
(R_1)	平均	22.6%	15.6%
	最大	66.9%	53.6%
(A_W)	平均	12.0%	10.3%
	最大	53.5%	47.7%
(A_R)	平均	13.2%	10.7%
	最大	52.4%	45.5%

最後に，各モデルにおける最大および平均サイクル削減率を，**表 6** に示す．この結果より，RaR アクセスを制御することで Futile Stall を解決する本提案手法は，単純に競合発生時にいずれかのトランザクションをアボートさせるだけでは解決しえない性能低下を効率的に解決できていることが確認できた．

5.5　追加ハードウェアの実装コストとアクセスレイテンシ

本節では，提案手法の実現のために追加したハードウェアの実装コストとアクセスレイテンシについて考察する．

アドレス記憶数を 1 とするモデル (R_1) を 32 コア構成プロセッサ上に実装した場合を考えると，4 章でも述べたように Tgt-addr.，O-que. および R-res. に必要となる記憶容量は 1 コアあたりそれぞれ 64 bit，4 bit × 31，4 bit となり，32 コア総計でも 768 bytes とごく少量である．また，これら記憶装置以外に必要となる回路は，1 コアあたり，アドレス一致比較のための比較器 1 つと，O-que. のキュー操作のための回路のみであり，非常に少ないハードウェアコストで実装可能であることが分かる．

次に，これら追加ハードウェアに対するアクセスレイテンシによるアクセスオーバヘッドが性能に及ぼす影響について考察する．このオーバヘッドを算出するために，各ベンチマークプログラムの 31 スレッド実行時において各追加ユニットがアクセスされた回数を計測した．計測結果を**表 7**，**表 8** および**表 9** に示す．

これら各ユニットへのアクセス回数と，各ユニットのアクセスレイテンシを乗じたものの総和が，追加ハードウェ

表 7　(R_1) における Tgt-addr. へのアクセス回数
Table 7　Access count of Tgt-addr. with (R_1).

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

表 8　(R_1) における O-que. へのアクセス回数
Table 8　Access count of O-que. with (R_1).

GEMS		SPLASH-2		STAMP	
Btree	21,137	Barnes	417	Kmeans	270
Contention	130	Cholesky	3,751	Vacation	7
Deque	3,210	Radiosity	2,991		
Prioque	3,022	Raytrace	38,524		

表 9　(R_1) における R-res. へのアクセス回数
Table 9　Access count of R-res. with (R_1).

GEMS		SPLASH-2		STAMP	
Btree	22,113	Barnes	448	Kmeans	324
Contention	152	Cholesky	5,888	Vacation	10
Deque	3,303	Radiosity	4,052		
Prioque	3,232	Raytrace	39,456		

アのアクセスオーバヘッドとなる．ここで Tgt-addr. は，アドレスの記憶数を 1 とした場合，単純なレジスタで構成できるため，アドレスの保存および一致比較はそれぞれ 1 cycle 程度で行えると考えられる．一方で O-que. に対する 1 操作は，1 度の 4 bit シフトと 1 度の論理演算で行えるため 2 cycle 程度，R-res. に対する 1 操作は Tgt-addr. 同様コア番号の登録および一致比較にいずれも 1 cycle 程度を要すると考えられる．

これらの各ユニットに想定されるアクセスレイテンシおよび計測したアクセス回数から，各ベンチマークプログラムにおけるアクセスオーバヘッドが総実行サイクル数に占める割合を算出したところ，最も割合の大きかった Raytrace においても 0.2%程度であった．このことから，提案手法のために追加したハードウェアのアクセスオーバヘッドが性能に与える影響はごくわずかなものであることが確認できた．

6. おわりに

本稿では，Read→Write の順序でアクセスされるアドレスへの Read-after-Read アクセスを制御し，このアクセスに関わるスレッドの実行を逐次化するスレッドスケジューリング手法を提案した．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ボートを抑制した．また，提案手法を実現するために必要となるハードウェアのコストはいずれも非常に小さいことが分かった．

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 31 スレッド並列実行時において最大 66.9%，平均 22.6%，16 スレッド並列実行時において最大 53.6%，平均 15.5%の実行サイクル数が削減されることを確認した．

なお本稿で提案したモデルでは，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手法を探ってゆく必要がある．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も有効な処理を割り当て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. and Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. 20th Annual Int'l Symp. on Computer Architecture*, pp.289–300 (1993).

[2] Moravan, M.J., Bobba, J., Moore, K.E., Yen, L., Hill, M.D., Liblit, B., Swift, M.M. and Wood, D.A.: 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, J.E.B. and Hosking, A.L.: Nested Transactional Memory: Model and Preliminary Architecture Sketches, *Science of Computer Programming*, Vol.63, No.2, pp.186–201 (2006).

[4] Lupon, M., Magklis, G. and González, A.: A Dynamically Adaptable Hardware Transactional Memory, *Proc. 43rd Annual IEEE/ACM Microarchitecture* (*MICRO*), pp.27–38 (2010).

[5] Shriraman, A., Dwarkadas, S. and Scott, M.L.: Flexible Decoupled Transactional Memory Support, *Proc. 35th Annual Int'l Symp. on Computer Architecture* (*ISCA*), 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 Int'l Conf. on High-Performance Computer Architecture* (*HPCA-17*), 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. 20th Euromicro Int'l Conf. on Parallel, Distributed and Network-Based Processing* (*PDP'12*), pp.221–228 (2012).

[10] Goodman, J.R., Vernon, M.K. and Woest, P.J.: Efficient Synchronization Primitives for Large-Scale Cache-

Coherent Multiprocessors, *Proc. 3rd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS III*), pp.64–75 (1989).
[11] Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D. and Wood, D.A.: LogTM: Log-based Transactional Memory, *Proc. 12th Int'l Symp. on High-Performance Computer Architecture*, pp.254–265 (2006).
[12] Bobba, J., Moore, K.E., Volos, H., Yen, L., Hill, M.D., Swift, M.M. and Wood, D.A.: Performance Pathologies in Hardware Transactional Memory, *Proc. 34th Annual Int'l Symp. on Computer Architecture* (*ISCA'07*), pp.81–91 (2007).
[13] 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).
[14] 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).
[15] Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd Annual Int'l Symp. on Computer Architecture* (*ISCA'95*), pp.24–36 (1995).
[16] 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*), pp.35–46 (2008).
[17] Gramoli, V. and Guerraoui, R.: Transactions — stamp (2011), available from ⟨http://lpdserver.epfl.ch/transactions/wiki/doku.php?id=stamp⟩.
[18] 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).

橋本 高志良　（学生会員）

1990 年生．2013 年名古屋工業大学工学部情報工学科卒業．現在，同大学大学院工学研究科創成シミュレーション工学専攻博士前期課程在籍．計算機アーキテクチャ，並列処理等に興味を持つ．

堀場 匠一朗　（学生会員）

1989 年生．2012 年名古屋工業大学工学部情報工学科卒業．現在，同大学大学院工学研究科創成シミュレーション工学専攻博士前期課程在籍．計算機アーキテクチャ，マルチコア・プロセッサ等に興味を持つ．

江藤 正通

1987 年生．2011 年名古屋工業大学工学部情報工学科卒業．2013 年同大学大学院工学研究科創成シミュレーション工学専攻博士前期課程修了．同年東海旅客鉄道（株）入社．計算機アーキテクチャ，並列処理等に興味を持つ．

津邑 公暁　（正会員）

1973 年生．1996 年京都大学工学部情報工学科卒業．1998 年同大学大学院工学研究科情報工学専攻修士課程修了．2001 年同大学院情報学研究科博士後期課程学修認定退学．同年同大学院助手．2004 年豊橋技術科学大学工学部助手．2006 年名古屋工業大学大学院工学研究科助教授．2007 年同准教授．博士（情報学）．プロセッサアーキテクチャ，並列処理，脳型情報処理等に関する研究に従事．ACM，IEEE-CS．電子情報通信学会各会員．

松尾 啓志　（正会員）

1960 年生．1985 年名古屋工業大学大学院修士課程修了．1989 年同大学院博士後期課程修了．同年同大学電気情報工学科助手．1993 年同講師．1995 年同助教授．2003 年同教授．2004 年同大学情報工学科教授．工学博士．計算機工学，分散協調システムに関する研究に従事．IEEE，人工知能学会，電子情報通信学会各会員．