卒業研究論文

アドレス情報を利用した
並列度の局所的低減による
ハードウェアトランザクショナルメモリの高速化

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学 工学部 情報工学科
平成 21 年度入学　21115118 番

橋本 高志良

平成 25 年 2 月 12 日

アドレス情報を利用した並列度の局所的低減による 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

内容梗概

マルチコア環境における並列プログラミングでは，共有リソースへのアクセスの調停にロックが広く用いられている．しかし，ロックにはデッドロックや並列性の低下が引き起こされるといった問題があるため、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，ロックを使用しない並行性制御機構としてトランザクショナル・メモリ（TM）が提案されている．TM は，データベースにおけるトランザクション処理をメモリアクセスに適用したものであり，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することで，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ことができ，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，同一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）．トランザクショナル・メモリ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（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，トランザクション操作のためのオーバヘッドを軽減している．この HTM では，ある共有リソースに対する Read-after-Read（RaR）アクセスは投機実行の継続が可能であるため，一般的には競合として検出されない．しかし，そのような RaR アクセスに関与したトランザクションの一方が結果的にアボートされると，その過程において発生したストールが完全に無駄となり，HTM の全体性能を大きく低下させてしまう場合がある．

そこで本論文では，このような問題を引き起こしうる RaR アクセスを検出し，その RaR アクセスに関わるトランザクションをあえて逐次実行する手法を提案する．提案手法の有効性を検証するため，HTM の実装の 1 つである 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 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，無駄なストールの抑制により最大 66.9%，平均 22.6%の実行サイクル数を削減した．

アドレス情報を利用した並列度の局所的低減による
ハードウェアトランザクショナルメモリの高速化

目次

1　はじめに　1

2　TM の概要　2

2.1　TM の基本概念 2

2.2　データのバージョン管理 4

2.3　競合の検出と解決 6

3　関連研究　9

4　競合アドレス情報に基づく並列度の低減　10

4.1　Read-after-Read アクセスに起因する問題 11

4.2　Futile Stall 抑制手法 12

4.2.1　Read-after-Read アクセスの検出 12

4.2.2　待機スレッドの再開順序制御 14

5　アドレス情報の利用と再開順序制御　16

5.1　拡張したハードウェアの構成 16

5.2　Read-after-Read アクセス検出の動作モデル 17

5.2.1　RaR addr. へのアドレス登録 18

5.2.2　RaR addr. を利用した RaR アクセス検出 19

5.2.3　RaR addr. のハードウェアコスト 20

5.3　再開順序制御の動作モデル 21

5.3.1　トランザクションの再開順序制御 21

5.3.2　再開順序制御のためのハードウェアコスト 23

6　評価結果と考察　24

6.1　評価環境 24

6.2　評価結果 24

6.3　考察 28

6.4　RaR addr. の参照コスト 30

7　おわりに　31

著者発表論文 **32**

参考文献 **32**

1 はじめに

これまでのプロセッサ高速化技術は，スーパスカラやSIMDのような命令レベル並列性（Instruction-Level Parallelism: ILP）に基づいた多くの高速化手法や，集積回路の微細化による高クロック化の実現によって支えられてきた．しかし，プログラム中で抽出できるILPには限界があり，また消費電力や配線遅延の相対的な増大から，単一コアにおける動作クロック周波数の向上も困難となりつつある．

これらの流れを受け，単一チップ上に複数のプロセッサ・コアを搭載したマルチコア・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，これまで単一コアで実行していたタスクを複数のプロセッサ・コアが分担することで，単一コアで実行した場合よりもスループットを向上させることができる．このようなマルチコア環境における並列プログラミングでは，複数のプロセッサ・コア間で単一アドレス空間を共有する，共有メモリ型並列プログラミングが一般的である．このプログラミングモデルでは，共有リソースへのアクセスを調停する必要があり，その調停を行う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，ロック操作のオーバヘッド増大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ことは難しいため，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，ロックを使用し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は，データベースにおけるトランザクション処理をメモリアクセスに適用したものであり，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することで，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ことができ，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，同一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）．トランザクショナル・メモリ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，トランザクション操作のためのオーバヘッドを軽減している．このような利点からHTMは現在大きな注目を集めており，2012年にIBM社が開発・発表し

たスーパーコンピュータであるBlueGene/Q [2] や，2013年にIntel社から出荷予定のHaswell [3] などに実装が始まっている．

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アクセスが発生した場合，競合とはならず，トランザクション実行は継続される．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されると，その過程において発生したストールが完全に無駄となり，HTMの全体性能を大きく低下させてしまう場合がある．そこで本論文では，このような問題を起こし得るRead-after-Readアクセスを検出し，そのようなトランザクション処理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．

以下，2章では本研究の対象であるTMおよびHTMの概要について説明し，3章では，本研究に関連する手法について説明する．4章でHTMの問題点及び提案手法について説明し，5章でその動作モデルについて説明する．6章で提案手法を評価し，7章で結論を述べる．

2　TMの概要

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（TM）および，それをハードウェアで実現したシステムであるハードウェア・トランザクショナル・メモリ（HTM）について述べる．

2.1　TM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，複数のプロセッサ・コアが単一アドレス空間を共有する．したがって，異なるプロセッサ・コアによる同一メモリアドレスに対するアクセスを調停する必要があり，その調停を行う機構として一般的にロックが用いられている．しかし，ロックを使用してアクセスを調停する場合，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や使用するロック変数自体が増加した場合，ロックの獲得・解放操作に要するオーバヘッドも増加し，性能が低下する可能性がある．さらに，プログラムごとに適切なロックの粒度を設定することは難しい．例えば，粗粒度なロックを用いる場合，プログラムの構築は容易であるがクリティカルセクションが大きくなるため並列性が損なわれる．これに対して細粒度なロックを用いる場合，並列性は向上するが大規模なプログラムであ

るほど設計が複雑になる．以上のような特徴は，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．

そこでロックを使用しない並行性制御機構である**トランザクショナル・メモリ（Transactional Memory: TM）**が提案されている．TM はデータベース上で行われるトランザクション処理をメモリアクセスに対して適用した手法である．TM では，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，これを投機的に実行することでロックを用いる場合よりも並列度を向上させている．このトランザクションは以下の 2 つの性質を満たす．

Serializability（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り，全てのスレッドにおいて同一の順序で観測される．

Atomicity（不可分性）：
トランザクションはその操作が完全に実行されるか，または全く実行されないかのいずれかでなければならず，各トランザクション内における操作はトランザクションの終了と同時に観測される．そのため，操作の途中経過が他のスレッドから観測されることはない．

以上の性質を保証するために，TM はトランザクション内のメモリアクセスを監視する．しかし，複数のトランザクション内で同一アドレスへのアクセスが確認された際に，これらのアクセスがトランザクションの性質を満たさない場合，**競合**として検出される．この操作を**競合検出（Conflict Detection）**という．このように競合が検出された場合，TM では一方のトランザクションの実行を停止することで競合を回避する．これを**ストール（stall）**という．さらに，複数のトランザクションがストールした状態でデッドロックが発生する可能性がある場合，TM では一方のトランザクションの途中結果を全て破棄する．この操作を**アボート（abort）**という．その後，トランザクションをアボートしたスレッドはトランザクション開始時点から処理を再実行する．これに対して，トランザクションが終了するまで競合が発生しなかった場合，トランザクション内で更新された全てのデータをメモリに反映させる．この操作を**コミット（commit）**という．なお，トランザクションの投機的実行では，上述のとおり途中結果が破棄される可能性があるため，トランザクション内で更新した値と更新前の両方の値を保持しておく必要がある．そこで TM では，トランザクション内で発生した Write アクセスにより更新したデータ，あるいは更新される前の古いデータを，そのアドレスとともに別領域に保持する．このようなデータの管理を**バージョン管理（Version**

Management）という．

TM は以上のように動作することで，競合が発生しない限りトランザクションを並列実行することができる．なお，TM で行われる競合検出やデータのバージョン管理などの操作はハードウェア上またはソフトウェア上に実装されている．ソフトウェア上に TM を実装したソフトウェア・トランザクショナル・メモリ (STM) [4] では，TM で行われる操作が全てソフトウェアによって実現されるため，特別なハードウェア拡張は必要ないが，ソフトウェア処理のためのオーバヘッドが大きい．これに対し，ハードウェア上に実装された TM はハードウェア・トランザクショナル・メモリ (HTM) と呼ばれる．一般的に HTM では，競合を検出および解決する機構をハードウェアによってサポートしているため，STM に比べて速度性能が高い．

2.2　データのバージョン管理

さて，上述のとおり TM におけるトランザクションの投機的実行では，実行結果が破棄される場合に備えてデータのバージョン管理が行われる．このバージョン管理におけるデータの管理方式は以下の 2 つの方式に大別される．

Eager Version Management:

更新前の古い値を別領域にバックアップし，新しい値をメモリに上書きする．コミットはバックアップを破棄するだけなので高速に行えるが，アボート時に，バックアップされた値をメモリに書き戻す必要がある．

Lazy Version Management:

更新前の古い値をメモリに残し，新しい値を別領域に保持する．更新前の値がメモリに残っているためアボートは高速に行うことができるが，コミット時に，更新した値をメモリに反映させる必要がある．

前者の Eager 方式は，必ず実行されるコミットを高速に行い，必ずしも発生するとは限らないアボートに処理コストを払う方式である．そのため、アボートが頻発するようなプログラムでは処理が遅くなる可能性がある．しかし，Lazy 方式におけるコミットのオーバヘッドは削減の余地がほぼ無いのに対し，Eager 方式におけるアボートのオーバヘッドは削減の余地が大いにある．これは効率的なスレッドスケジューリングを行うことでアボートや競合の発生自体を抑制できるためである．よって本論文では，Eager 方式を研究対象とする．

ここで，この Eager 方式におけるバージョン管理の動作を**図 1** と**図 2** を用いて説明する．これらの図は，トランザクションが投機的に実行される様子を示しており，*Memory*

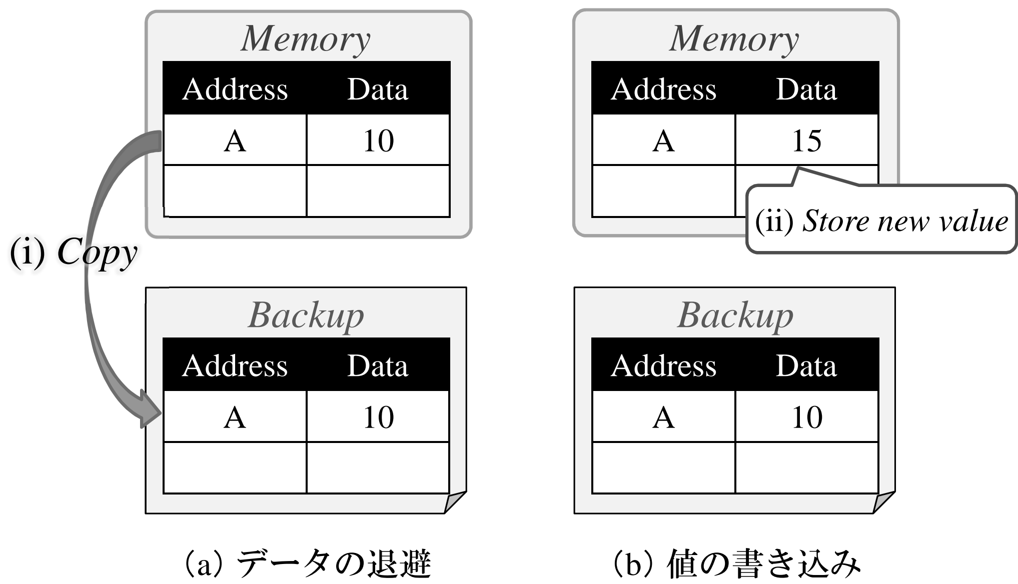


図 1: Write アクセス時のバージョン管理

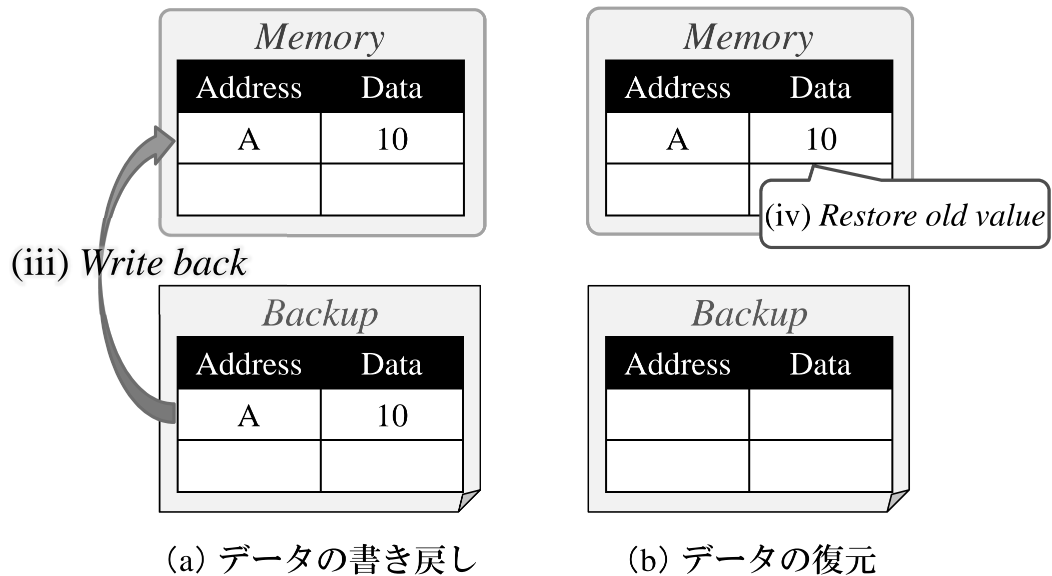


図 2: アボート実行時のバージョン管理

および *Backup* はそれぞれメモリ，バックアップ領域を表す．まず，メモリアドレス A に値 10 が格納されている状態で，トランザクションの実行が開始されたとする．この状態からトランザクションの実行が進み，アドレス A に値 15 が書き込まれるとすると，図 1(a) に示すように Write アクセスの対象アドレス A と，書き換え前の値である 10 がメモリからバックアップ領域に退避され (i)，図 1(b) のように書き込みの結果で

ある 15 がメモリに上書きされる (ii).

次に図 1(b) の状態からさらに実行が進み，投機実行が成功した場合には，トランザクションがコミットされる．このとき，書き込みの結果である値 15 は既にメモリに保持されているため，バックアップ領域の内容を破棄するだけでコミットを実現できる．一方で投機実行が失敗した場合，実行トランザクションはアボートされる．アボート時の操作では，図 2(a) に示すように，バックアップ領域に退避されたデータが元のメモリアドレスに書き戻される (iii)．これにより，図 2(b) のようにトランザクション開始時点のメモリ状態を復元することができる (iv)．また，アボート後にトランザクションを再実行するためには，メモリ状態と同様にレジスタをトランザクション開始時の状態に戻す必要がある．これを実現するために HTM では，トランザクション開始時にその時点におけるレジスタの状態を取得し，その状態をバックアップ領域に保存しておく．そして，アボート時にバックアップ領域を参照し，トランザクションの開始時点のレジスタ状態を復元する．

2.3　競合の検出と解決

トランザクションの Atomicity を保つためには，ある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内のメモリアクセスとの間に競合が発生するかどうかを検査する必要がある．そのため，トランザクション内でどのメモリアドレスがアクセスされたかを記憶しておかなければならない．これを実現するために，HTM では各キャッシュライン上に **read/write** ビット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 Read または Write アクセスが発生すると，アクセスされたキャッシュラインに対応する read または write ビットがセットされる．そして，各ビットはトランザクションのコミット，およびアボート時にクリアされる．

これらのビットを操作するため，HTM ではキャッシュの一貫性を保持するプロトコルであるディレクトリベース [5] の Illinois プロトコル [6] を拡張している．このプロトコルでは，あるスレッドがメモリアクセスする場合，キャッシュ・コヒーレンス・リクエスト（以下，リクエスト）を各スレッドに送信する．拡張したプロトコルにおいて，各スレッドはリクエストを受信する際，キャッシュラインの状態を変更する前に，当該ラインに追加された read または write ビットを参照する．これにより，各スレッドが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際に，そのアドレスが他のスレッドの実行するトランザクション内で既にアクセスされているかを検査することができる．この検査では，以下の 3 パターンのメモリアクセスを競合と

して検出する．

Read after Write（RaW）：

あるトランザクション内で Write 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 Read アクセスされるパターン．

Write after Read（WaR）：

あるトランザクション内で Read 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 Write アクセスされるパターン．

Write after Write（WaW）：

あるトランザクション内で Write 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 Write アクセスされるパターン．

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．これにより，*NACK* を受信したスレッドは自身のアクセスによって競合が発生したことを知ることができる．ここで，実際に *NACK* および *ACK* を送信するのはキャッシュディレクトリであるが，本論文では便宜的に，先行して当該アドレスにアクセスしたスレッドがこれらを送信するものとして説明する．なお，Read after Read（RaR）アクセスはデータの一貫性に影響を与えないため，一般的に競合として検出されない．

また，この競合検出方式は競合検査のタイミングによって以下の 2 つに大別される．

Eager Conflict Detection:

トランザクション内でメモリアクセスが発生した時点で，そのアクセスに関する競合が存在しないか検査する．

Lazy Conflict Detection:

トランザクションをコミットしようとした時点で，そのトランザクション内で行われた全てのアクセスに関して競合が発生していないか検査する．

Lazy 方式では，トランザクション内で競合が発生してから検出されるまでの時間が Eager 方式に比べて長くなり，無駄な処理が増大して実行効率が悪くなる．これより，HTM では競合検出方式として Eager 方式が採用されている場合が多い．なお，一般的に Eager 方式では競合が発生した場合に競合相手のトランザクションがコミットもしくはアボートされるまで実行トランザクションをストールさせる．そして，実行トランザクションをストールさせたスレッドは同じアドレスに対するリクエストを送信し続ける．その後，競合相手のスレッドが実行トランザクションを終了した場合，そのスレッドから *ACK* が返信されるため，実行トランザクションをストールさせていた

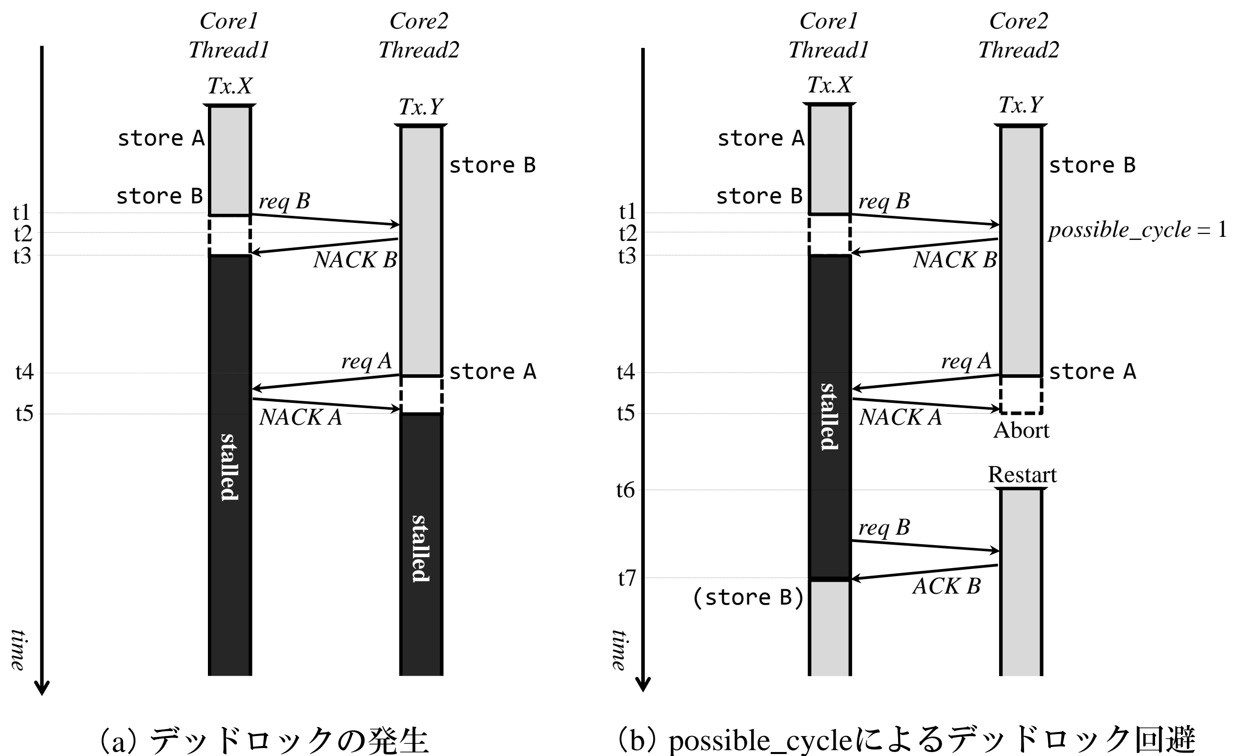


図 3: HTM におけるトランザクションの競合解決

スレッドは競合相手のトランザクションが終了したことを検知し，実行を再開できる．

しかし**図 3**(a) に示すように，複数のスレッドが実行トランザクションをストールさせるとデッドロック状態に陥る場合がある．この例では，*Core1* と *Core2* という 2 つのコア上で *Thread1* および *Thread2* が，それぞれトランザクション *Tx.X* および *Tx.Y* を投機的に実行している．まず *Thread1* が *Tx.X* の実行を開始した後に，*Thread2* が *Tx.Y* の実行を開始しており，*Thread1* が `store A` を，そして *Thread2* が `store B` を実行済である場合を考える．次に，*Thread1* は `store B` を実行しようとし，*Thread2* へアドレス B に対するリクエストである *req B* を送信する（時刻 t1）．この *req B* を受信した *Thread2* は，既に当該アドレス B にアクセス済であるため競合の発生を検知し，*Thread1* へ *NACK B* を返信する（t2）．*Thread1* は *NACK B* を受信すると，自身の *Tx.X* をストールさせる（t3）．なお，図中では省略しているが，*Thread1* はアクセスの許可を受けるまで *Thread2* に対して定期的にリクエストを送信し続ける．この後 *Thread2* が `store A` を実行しようとすると（t4），*Thread1* と *Thread2* との間で競合が発生するため，*Thread2* は自身の実行トランザクションである *Tx.Y* をストールさせる（t5）．このように，複数のスレッドが自身の実行トランザクションをストール

させ，競合したトランザクションの終了を互いに待ち続けると，デッドロック状態に陥ってしまう．

このようなデッドロック状態を回避するために，一般的なHTMではTransactional Lock Removal [7] に倣った方法を採用している．具体的には，デッドロックを発生させうるようなトランザクションの開始時刻が，競合相手のトランザクションよりも遅い場合に，そのトランザクションをアボートする．これは，各プロセッサ·コアに *possible_cycle* と呼ばれるフラグを保持させることで実現している．ここで，*possible_cycle* フラグによりデッドロックを回避する例を図3(b) に示す．*Thread2* は *Thread1* へ *NACK* を返信する際，*Thread1* が自身よりも早くトランザクションを開始しているため，*possible_cycle* フラグをセットする（t2）．そして，*possible_cycle* フラグをセットしたスレッドは，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* を受信した場合，デッドロックの発生を防ぐために自身の実行トランザクションをアボートする（t5）．このように *possible_cycle* フラグを利用することで，開始時刻の遅いトランザクションをアボートの対象として選択する．そして，実行トランザクションをアボートした *Thread2* はトランザクション開始時点のメモリ状態およびレジスタ状態を復元し，*Tx.Y* を再実行する（t6）．また，*Thread2* が実行トランザクションをアボートしたことで *Thread1* はアドレスBにアクセスできるようになるため，自身の実行トランザクションである *Tx.X* をストール状態から復帰させる（t7）．

3　関連研究

実行トランザクションをアボートした後にそのトランザクションを途中から再実行することで，再実行コストを抑える部分ロールバックに関する研究 [8–10] や，アプリケーションの振る舞いによって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 [11–13] など数多くのHTMに関する研究が行われてきた．しかし前者の研究では，競合しやすいアドレスに対するアクセスの直前にロールバックしてしまうため，競合がより再発しやすくなるという問題がある．一方，後者の研究ではバージョン管理と競合検出の方式を変更する際のオーバヘッドや，実装に必要なハードウェアコストについて詳細に評価していないため，実用的な手法ではない．また，これらの方式を変更せずともスケジューリングの改良次第で競合の発生を抑制できる可能性があるため，本研究ではスレッドスケジューリングに着目する．このスレッドスケジューリングについては，これまで主に以下に示す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 [14] 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ようなアプリケーションの実行を高速化する手法を提案した．この手法では既存の HTM に対して最大で 97%の実行速度向上を達成していたが，速度向上したプログラムはごく一部であり，その他のプログラムでは速度向上がなされていなかった．一方，Geoffrey ら [15] 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，この Similarity 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的に実行することで競合を抑制する手法を提案した．しかし，この手法は性能評価において関連手法のみを比較対象としているため，既存の HTM に対してどの程度速度性能が向上したのかが明確に示されていない．また，Akpinar ら [16] 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，既存の HTM に対して最大で 15%の速度向上を達成していた．しかし Akpinar らは，HTM においてボトルネックとなりうる競合パターンの発生状況の分析に重点を置いており，これらに対する解決手法はごく単純なものしか提案していない．

もう一方の方向性からの改良として，Gaona ら [17] 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．しかし，この手法では，既存の HTM に対して最大でも 10%程度の消費電力しか削減されておらず，実行速度も既存の HTM とほぼ同程度の性能にとどまっている．

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．一方本研究では，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．

4　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 において競合として検出されない RaR アクセスが原因となって発生する問題について述べ，これを抑制する手法を提案する．

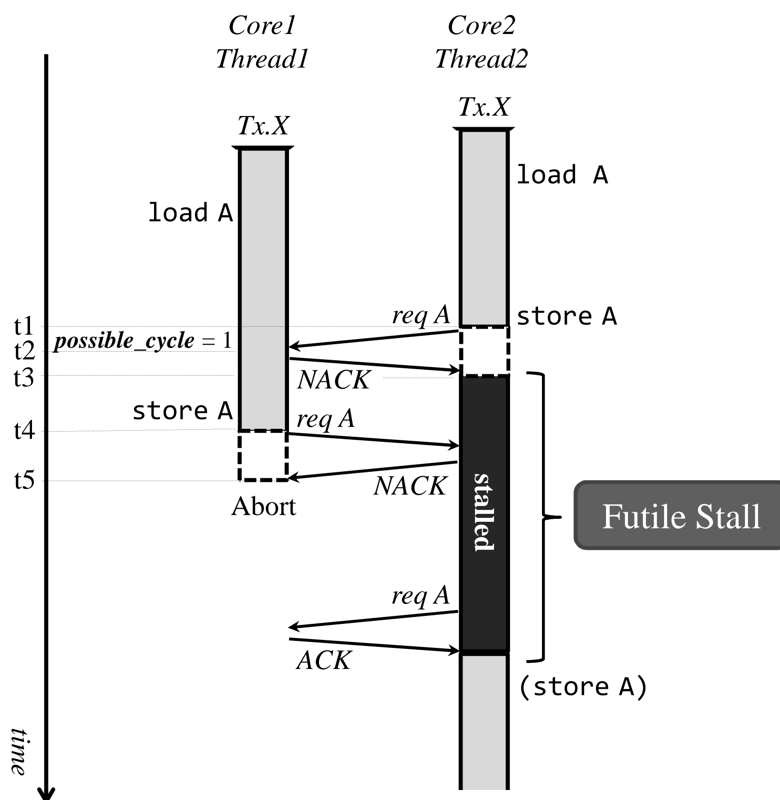


図 4: 2 スレッド並列実行時における Futile Stall の発生

4.1　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．具体的な例として，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ある．このような，ある共有変数に対して Write アクセスに先立って Read アクセスするようなトランザクション処理が，複数のスレッドによって並列実行される場合，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 アクセスにより競合が発生してしまうことになる．

図 4 は，上記の処理を含む共通のトランザクション *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と（時刻 t1），競合の発生により，*Thread1* は *Thread2* へ *NACK* を返信するとともに，自身の *possible_cycle* フラグをセットする（t2）．そして，*Thread2* は自身の *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* は既に当該アドレス `A` にアクセス済

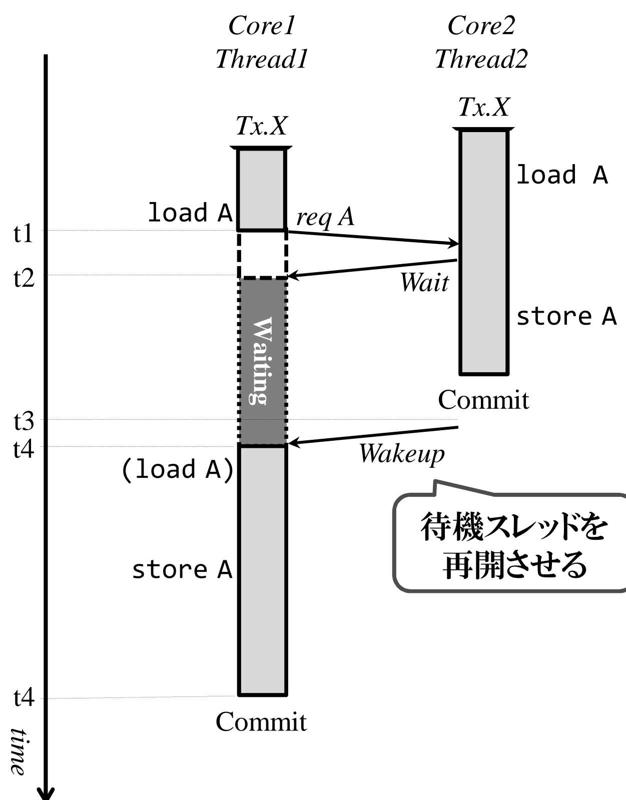


図 5: 2 スレッド並列実行時における Futile Stall の抑制

であるため競合を検出し，*Thread1* へ *NACK* を返信する．この時，*possible_cycle* フラグがセットされている状態で，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，*Thread1* は自身の *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 *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は **Futile Stall** 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．

4.2 Futile Stall 抑制手法

本節では，前節に示したような Futile Stall を抑制して HTM の性能低下を防ぐ手法を提案し，その動作について説明する．

4.2.1 Read-after-Read アクセスの検出

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，RaRアクセスであるか否かを検出することで，そのような状況を回避する．ここで，Futile Stallを抑制する手法を適用した例を**図5**に示す．まず，*Thread2*が`load A`を実行した後，*Thread1*が`load A`を実行しようとした場合（時刻t1），*Thread2*はRaRアクセスを検出し，*Thread1*へ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シ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（t2），*Thread1*のトランザクション実行は待機させられることになるため，*Thread2*はアドレス`A`にWriteアクセスしたとしても，*Thread1*と競合することなくトランザクションの実行を進めることができる．その後，*Thread2*は自身のトランザクションをコミットした際，*Thread1*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状態の解除を実現する（t3）．この*Wakeup*メッセージを受信した*Thread1*はトランザクションの実行を再開し（t4），*Thread2*と競合することなく*Tx.X*をコミットする．以上のように動作させることで，Futile Stallによる無駄なサイクルを削減することができる．しかし，3者以上のスレッドが*Tx.X*を並列に実行する場合，複数の待機スレッドの実行が一斉に再開され，これらのスレッド間で競合が発生する可能性がある．

ここで，上述したFutile Stall抑制手法を，3つのスレッドが*Tx.X*を並列実行する場合に適用した際の動作を**図6**に示す．この例では（*Thread1*〜*3*）が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*Wait*リクエストを送信する（t3，t4）．その後，*Thread2*は自身のトランザクションをコミットした際，*Thread1*と*Thread3*に*Wakeup*メッセージを送信して，それぞれの実行を再開させる（t5）．しかしこの場合においては，単純に*Thread1*および*Thread3*へ同時に*Wakeup*メッセージを送信したのでは，これらのスレッド間で競合が発生する（t6，t7）．なお，簡略化のために図6中において，時刻t5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，発生した競合によって*Thread3*が*Tx.X*を結果としてアボートするため（t8），*Thread1*のストールが無駄なものとなってしまう．これを解決するために，RaRアクセスを検出した*Thread2*が，トランザクションの実行を待機している*Thread1*と*Thread3*の再開順序を制御する手法を併せて提案する．この待機スレッドの再開順序制御については次項で説明する．

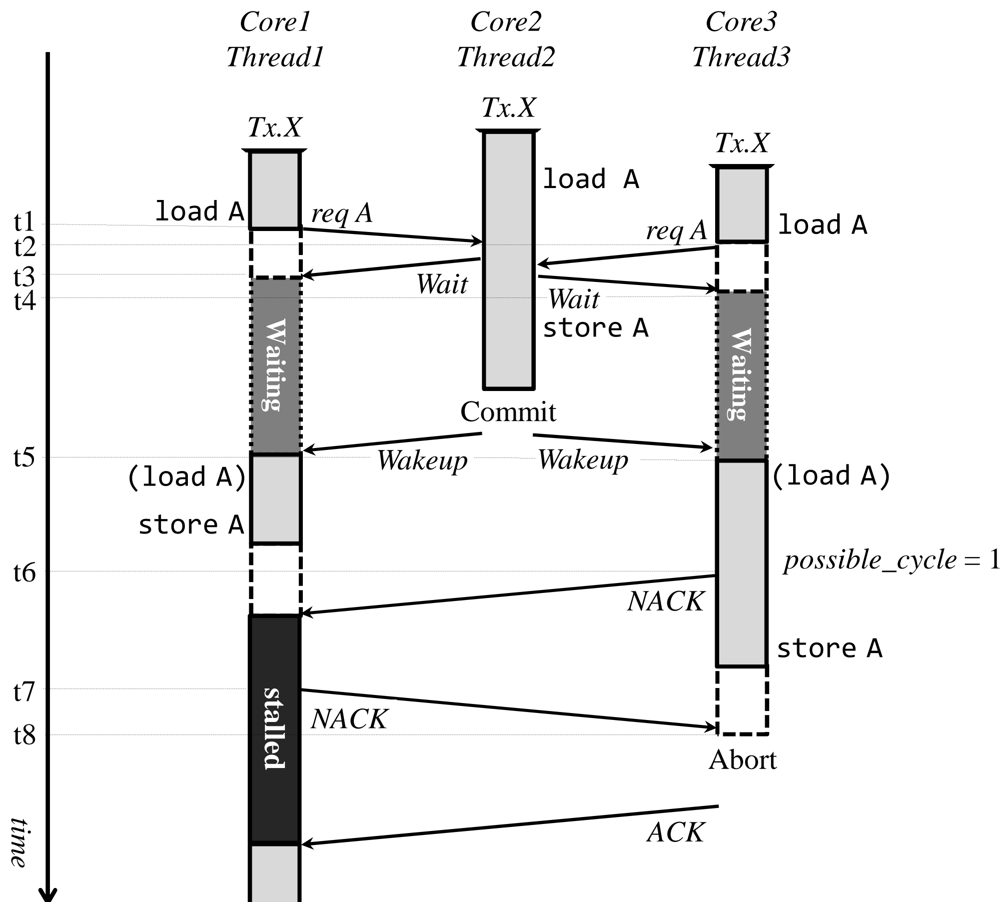


図 6: 3 スレッド並列実行時における Futile Stall 抑制手法の問題

4.2.2　待機スレッドの再開順序制御

図 6 における問題を解決するため，待機スレッドの再開順序を制御する必要がある．この再開順序制御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，受信した順を記憶しておき，自身の実行トランザクションをコミットした際にその順序で待機スレッドを *Wakeup* していくことで実現する．

ここで，再開順序制御を適用した場合の動作を**図 7** に示す．この例では図 6 と同様に，3 つのスレッド（*Thread1*〜*3*）が同一のトランザクション（*Tx.X*）を並列に実行している．まず *Thread2* が `load A` を実行した後，*Thread1* が `load A` を実行しようとした場合，*Thread2* は RaR アクセスを検出して *Thread1* へ *Wait* リクエストを送信すると同時に，*Thread1* から Read アクセスのための *req A* を受信したことを記憶する（時刻 t1）．続いて *Thread3* が Read アクセスを試みた場合，*Thread2* は *Thread1* と同様に RaR アクセスを検出し，*Thread3* へ *Wait* リクエストを送信する．そして，*Thread2*

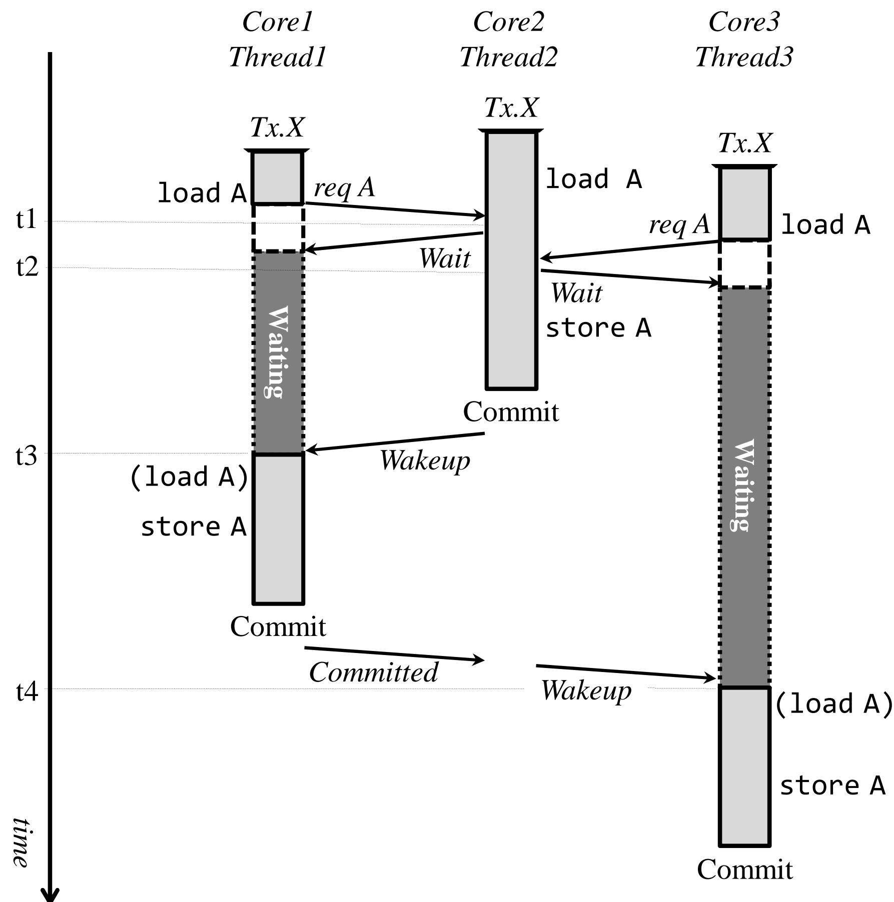


図 7: 待機スレッドの再開順序制御

は *Thread3* から Read アクセスのリクエストを受信することで *Thread1* → *Thread3* という順序で Read アクセスのリクエストを受信したことを記憶する（t2）．その後，*Thread2* が自身の実行トランザクションをコミットした際に，記憶した Read アクセス順序に基づいて再開順序を制御する．この場合では，*Thread3* より先に *Thread1* が Read アクセスを試みたため，*Thread2* はまず *Thread1* へ *Wakeup* メッセージを送信することで *Thread1* の実行を再開させる（t3）．実行を再開した *Thread1* は自身の *Tx.X* をコミットした際に，再開順序を制御している *Thread2* へ，コミットしたことを伝えるために *Committed* 通知を送信する．この *Committed* 通知を受信した *Thread2* は，*Thread1* がコミットしたことを検知し，続いて *Thread3* に対して *Wakeup* メッセージを送信することで *Thread3* の実行を再開させる（t4）．

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*〜*3*）が，再開順序制御に関与していないスレッドと競合する場合がある．例えば，*Thread1*

が再開順序制御に関与していないスレッドと競合して *Tx.X* をストールさせると，それに伴って *Thread3* の再開時刻も遅くなるため，再開順序制御の効率が悪くなってしまう．そこで本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，前者のスレッドが優先的に実行され，後者のスレッドが実行トランザクションをアボートすることで，再開順序制御の効率が悪化してしまうことを防ぐ．次章では，本手法の具体的な動作モデルおよび追加ハードウェアについて説明する．

5　アドレス情報の利用と再開順序制御

本章では，RaR アクセスの検出と待機スレッドの再開順序制御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説明する．

5.1　拡張したハードウェアの構成

Read/Write の順序でアクセスされるアドレス情報を利用した RaR アクセスの検出，および待機スレッドの再開順序制御を実現するため，既存の HTM を拡張して以下の 3 つのハードウェアを各コアに追加する．なお，コア数および最大同時実行スレッド数は n とする．拡張した HTM の構成を**図 8** に示す．

Register for RaR addresses（**RaR addr.**）:
各スレッドの実行トランザクションの中で 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．このキューには，Read/Write の順序でアクセス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．このレジスタ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自身のトランザクションをコミットした際に，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．

本手法では RaR アクセスを検出する必要があるため，RaR addr. というレジスタを追加する．各スレッドは，自身が Read/Write の順序でアクセスしたアドレスを RaR

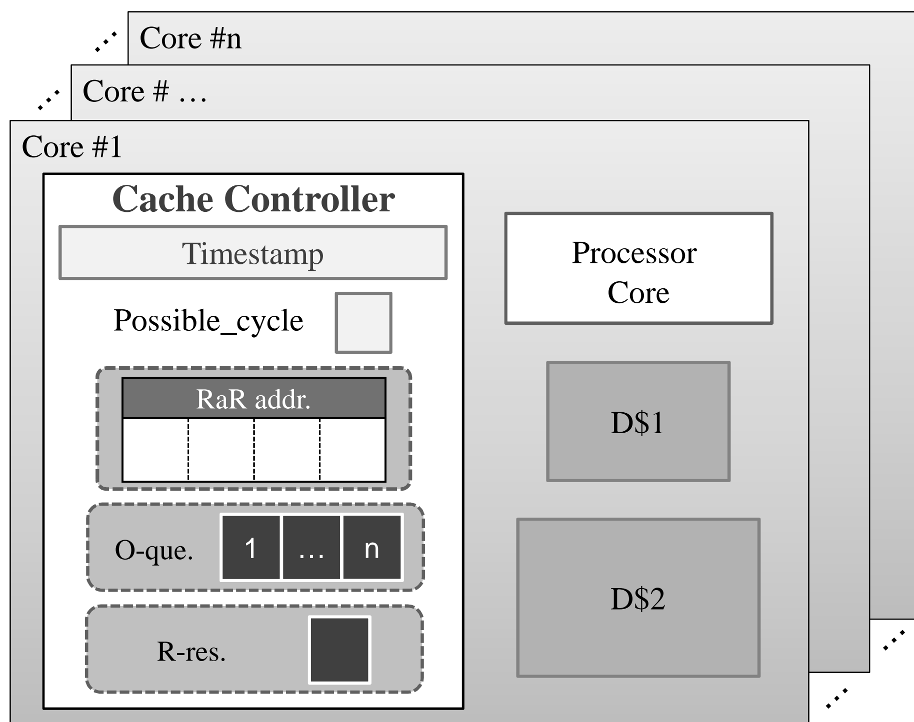


図 8: 拡張したハードウェアの構成

addr. に保持する．この RaR addr. 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. を参照することで RaR アクセスを検出する．さらに，実行を待機させたスレッドを順に再開させるために O-que. というキューを追加する．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，自身の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した後に，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，自身のトランザクションをコミットした際にその 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．

5.2 Read-after-Read アクセス検出の動作モデル

本節では，Read/Write の順序でアクセスされるアドレスの登録，そして RaR addr. に登録されたアドレスに基づいて RaR アクセスを検出する動作モデルを，前節で述べた拡張ハードウェアを踏まえて説明する．

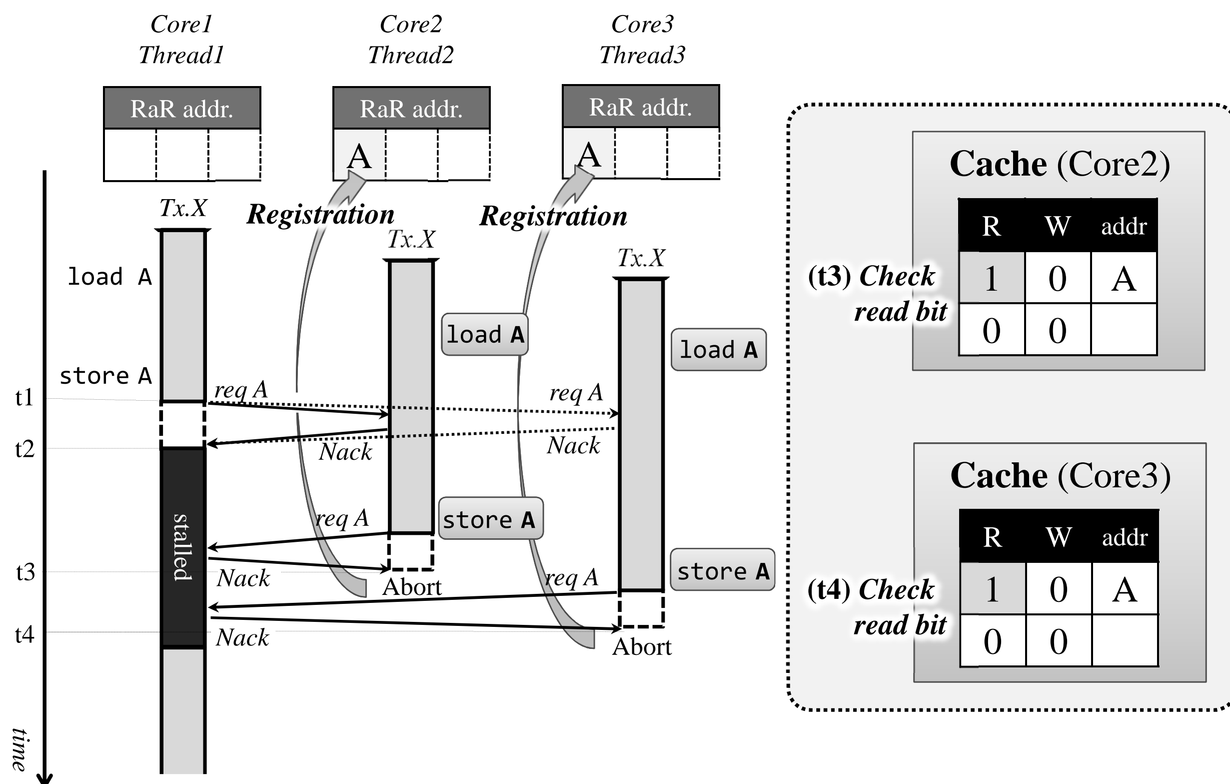


図 9: RaR アクセスを検出すべきアドレスの検知と RaR addr. への登録

5.2.1 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 9** を例に，追加した RaR addr. へアドレスを登録する動作について述べる．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際（時刻 t1），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* をストールさせる（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トランザクション *Tx.X* をアボートする．この時，*Thread2* と *Thread3* は Write アクセスを試みたアドレス `A` における自身の read ビットをチェックする（t3，t4）．この例では，*Thread2* および *Thread3* の当該アドレス `A` おける read ビットがセットされており，これらのスレッドは自身が 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，このアドレスをそれぞれの RaR addr. に登録する．

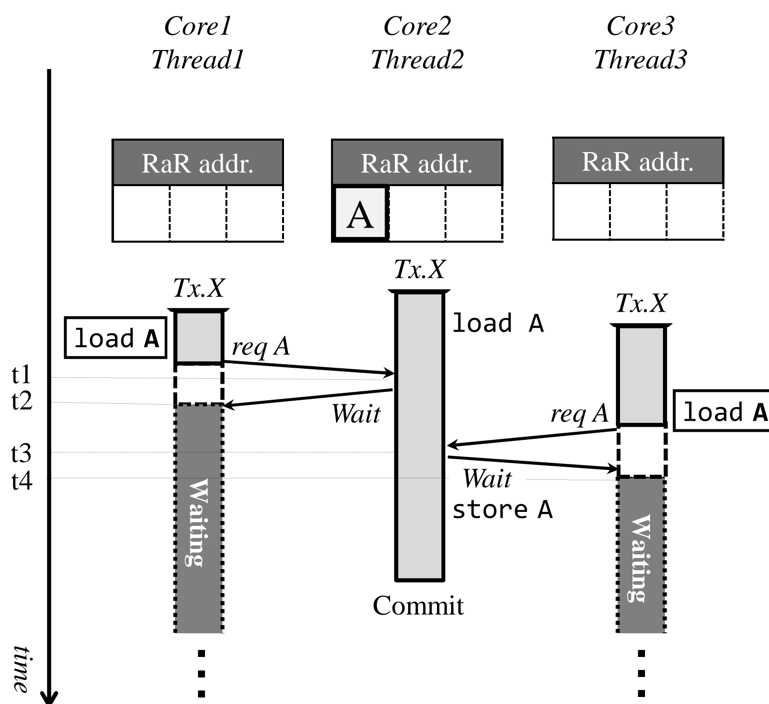
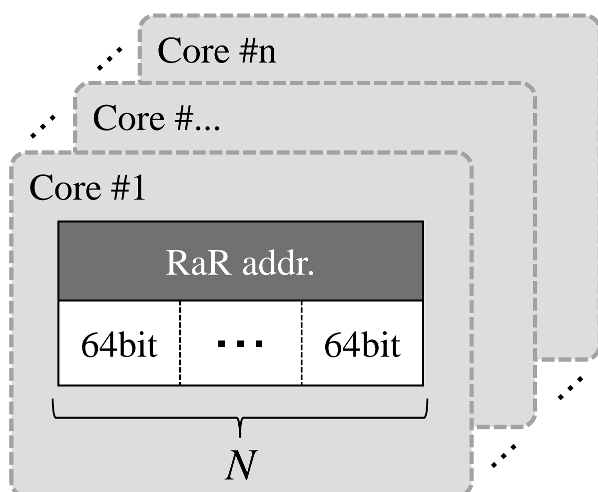


図 10: RaR addr. を利用した RaR アクセスの検出

5.2.2 RaR addr. を利用した RaR アクセス検出

5.2.1 項で述べた方法によって RaR addr. レジスタに登録されたアドレスを利用して，どのように RaR アクセスを検出するのかを**図 10** を用いて説明する．ここで，3 つのスレッド（*Thread1*〜*3*）はそれぞれ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 10 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` を実行し，*Thread1* が `load A` を実行しようとする．この時，*Thread1* は Read アクセスを試みるため *Thread2* へ *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` が登録済みのアドレスか否かを確認する．この時，*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 アクセスが，自身が過去に Read/Write の順序でアクセスしたアドレス `A` に対する Read アクセスであると分かる．したがって，*Thread2* は RaR アクセスを検出し，*Thread1* へ *Wait* リクエストを返信する．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



登録数	コスト / 32cores
$N = 1$	256 bytes
$N = 2$	512 bytes
$N = 4$	1024 bytes

図 11: RaR addr. の構成とハードウェアコスト

して実行を待機する（t4）．以上のように RaR addr. を利用することで，Read/Write の順序でアクセスされるアドレスにおける RaR アクセスを検出できる．

5.2.3　RaR addr. のハードウェアコスト

ここで，RaR アクセス検出のために追加した RaR addr. のハードウェアコストについて検討する．**図 11** に示すように，この 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/Wrire 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確保することは現実的ではない．したがって，このレジスタ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 のハードウェアコストで実現することが可能であり，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 と少量である．

なお，登録アドレス数を制限した場合，登録アドレスの管理はいくつかの選択肢をとり得るが，本論文では実装を簡略化するために単純な FIFO を採用する．例として，登録可能アドレス数を 4（$N = 4$）とした場合の様子を**図 12** に示す．図 12(a) に示すように，既にアドレス A が登録されている RaR addr. にアドレス B を新たに登録する場合，RaR addr. 全体を 64bit だけ右シフトしてアドレス B を登録する．また，図 12(b) のように，アドレスが最大まで登録された状態で新たなアドレスが新規に登録される場合，アドレスの右シフトにより登録済のアドレスの中で最も古いアドレス A が破棄されることとなる．この RaR addr. への登録アドレス数を増加させた場合，RaR アクセスをより正確に検出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

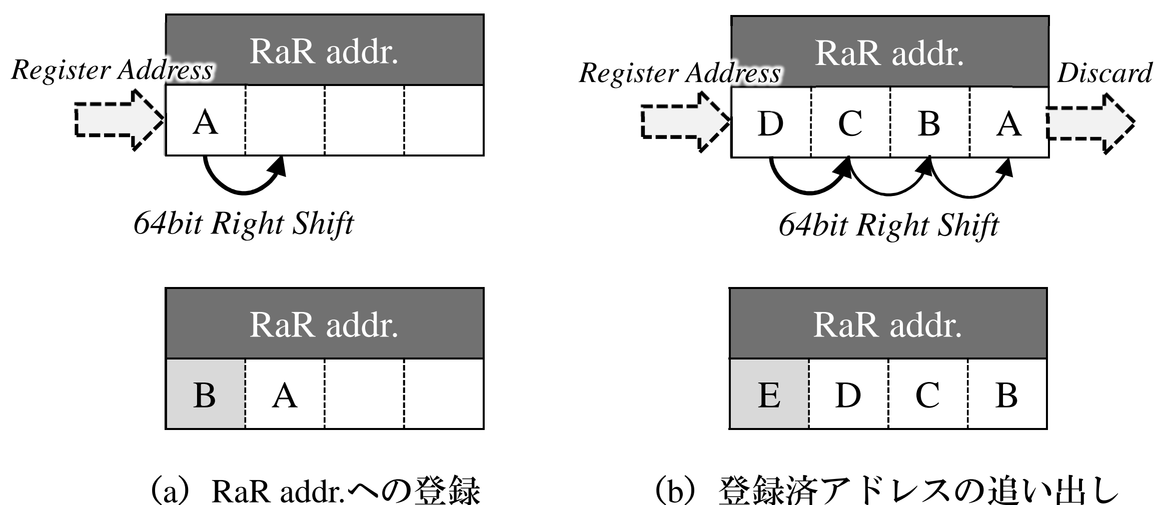


図 12: RaR addr. の操作

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 6 章で詳細に述べる．

5.3　再開順序制御の動作モデル

本節では，実行を待機させたスレッドの再開順序を制御する動作モデルについて説明する．

5.3.1　トランザクションの再開順序制御

5.2 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様子を**図 13** に示す．まず，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*）を開始する．なお，この例では既に *Thread2* の RaR addr. にアドレス `A` が登録済であるとする．

ここで *Thread2* が `load A` を実行した後，*Thread1* が `load A` を試みる場合，5.2.2 項で示したように *Thread2* は RaR addr. に登録されたアドレスに基づいて RaR アクセスを検出する．そして *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（時刻 t1）．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（t2）．そして，*Thread3* は R-res. に *Thread2* を実行するコア番号を格納する．

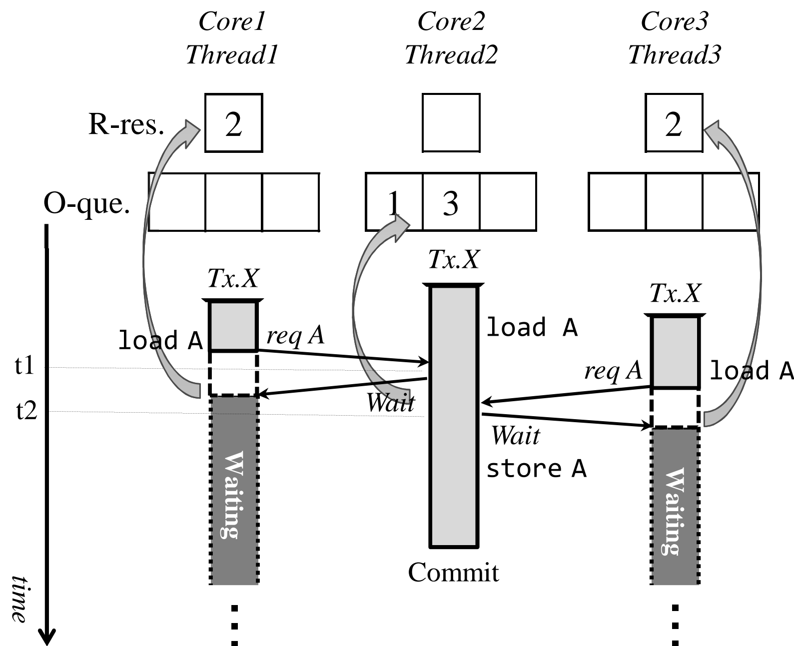


図 13: O-que. と R-res. の利用

次に，図 13 において O-que. および R-res. に格納し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動作を**図 14** に示す．図 14(a) は図 13 で *Thread2* が *Thread1* と *Thread3* の実行を待機させた後の状況を示している．ここで，*Thread2* はトランザクションをコミットした際（時刻 t1），自身の O-que. に格納されているコア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．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 *Thread1* 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．*Wakeup* メッセージを受信した *Thread1* は自身の *Tx.X* の実行を再開し，その後コミットに至る．*Tx.X* をコミットした *Thread1* は O-que. に格納されているコア番号 2 をチェックし，再開順序を制御しているスレッドが *Thread2* であると判断する．そして，*Thread1* は *Thread2* へ *Committed* 通知を送信することで，自身の *Tx.X* をコミットしたことを伝える（t2）．

その後，*Thread2* が再開順序の制御を続けていく様子を図 14(b) に示す．図 14(a) の動作によって，*Thread1* から *Committed* 通知を受信した *Thread2* は，再度自身の O-que. をチェックする（時刻 t1）．この例では，*Thread2* は自身の O-que. からコア番号 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

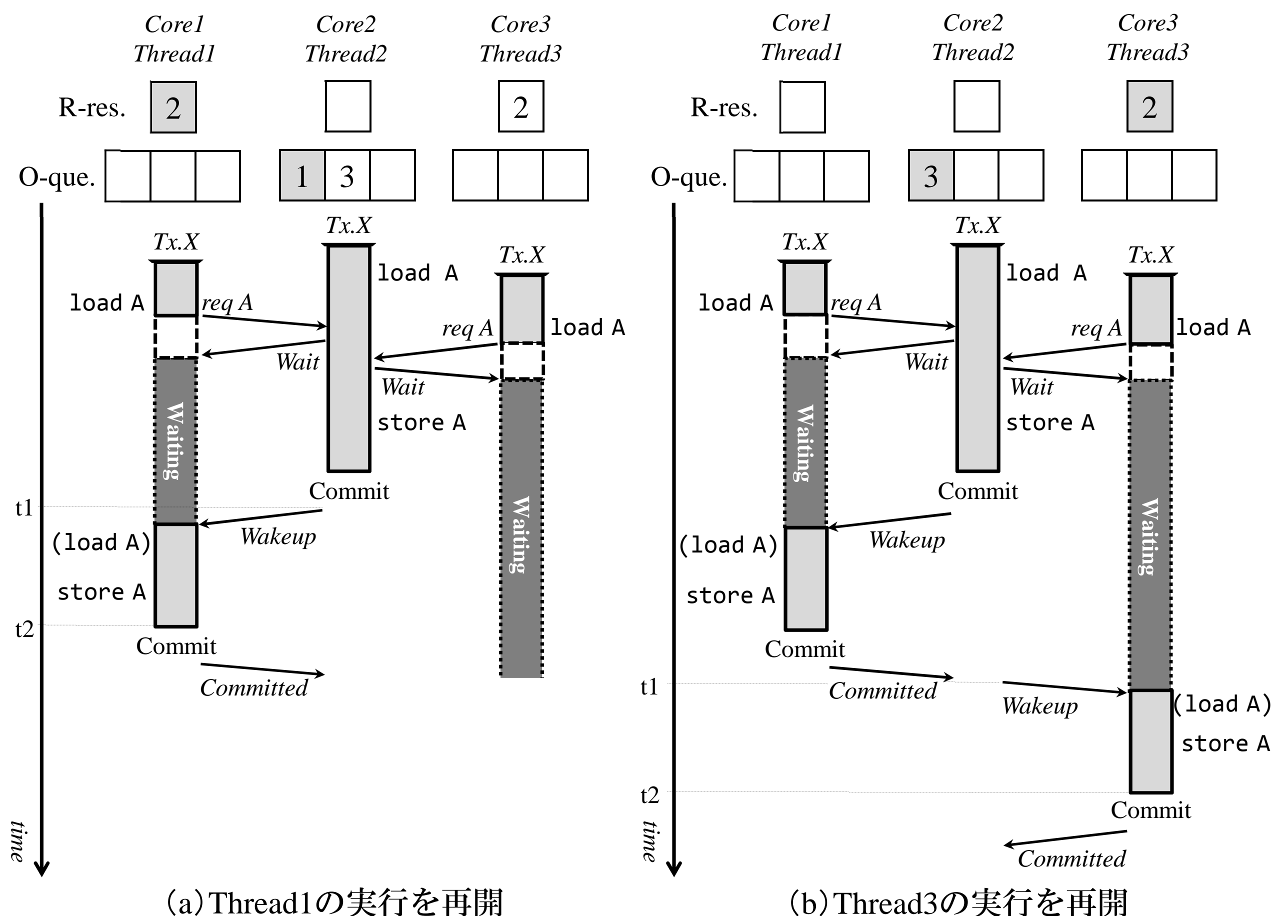


図 14: 再開順序制御すべきトランザクションの逐次実行

て *Tx.X* をコミットする．*Thread3* は *Tx.X* をコミットした後，自身の R-res. をチェックすることで，再開順序を制御しているスレッドが *Thread2* であると判断し，*Thread2* に対して *Committed* 通知を送信する（t2）．この *Committed* 通知を受信した *Thread2* は，もう一度自身の O-que. をチェックする．この時，この 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．

5.3.2　再開順序制御のためのハードウェアコスト

ここで，前項で説明した再開順序制御を実現するため，新たに追加したハードウェアのコストについて検討する．再開順序を制御するために追加した O-que. と R-res. のエントリには，**図 15** に示すようにいずれもコア番号が格納されるため，1 エントリあたり 4bit のコストとなる．また O-que. には，自身を除いたスレッドの数だけコア番号が格納されるため，32 コア構成のプロセッサにおいて 32 スレッドを並列実行する場

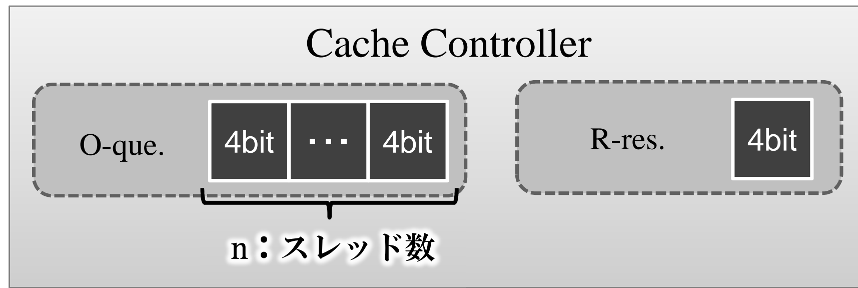


図 15: 再開順序制御のためのハードウェアコスト

合は最大で 31 個のエントリが必要となる．以上より，合計 32 個の 4bit のエントリが各コアに追加されることになるため，再開順序制御のためのハードウェアコストは全コア合計で 512bytes となる．

6 評価結果と考察

前章で述べた拡張を HTM の一実装である LogTM に実装し，評価を行った．本章ではその評価結果を示して考察し，その後 RaR addr. の参照コストについて述べる．

6.1 評価環境

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 3.0.31 [18] と GEMS 2.1.1 [19] 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タ構成を示す．評価対象のプログラムとしては GEMS 付属 microbench，SPLASH-2 [20]，および STAMP [21] から計 10 個を使用した．**表 2** に各ベンチマークプログラムの入力パラメタを示す．なお，32 コアの内の 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それぞれ 31 スレッドで実行した．また，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 [22] による，任意数のスレッド数での実行を可能にする改変を施した上で，31 スレッドで実行した．

6.2 評価結果

評価結果を**図 16** および**表 3** に示す．図 16 中の凡例はサイクル数の内訳を示しており，それぞれ以下のようになっている．

表 1: シミュレータ諸元

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

MagicWaiting: RaR アクセスの検出により実行を待機したサイクル数
Barrier: バリア同期に要したサイクル数
Stall: ストールに要したサイクル数
Backoff: アボート後に実行開始までランダム時間待つサイクル数
Aborting: アボートに要したサイクル数
Bad-trans: アボートされたトランザクションの実行サイクル数
Good-trans: コミットされたトランザクションの実行サイクル数
Non-trans: トランザクション外の実行サイクル数

図中では，各ベンチマークプログラムの評価結果が，各 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)　既存モデル（ベースライン）
(R_1)　アドレス登録数を 1 とした提案モデル
(R_2)　アドレス登録数を 2 とした提案モデル
(R_4)　アドレス登録数を 4 とした提案モデル

表 2: ベンチマークプログラムの入力パラメタ

GEMS	
Btree	priv-alloc-20pct
Contention	config 1
Deque	4096ops 128bkoff
Prioqueue	8192ops
SPLASH2	
Barnes	512
Cholesky	tk14.0
Radiosity	-p 31
Raytrace	teapot
STAMP	
Kmeans	random-n2048-d16-c16.txt
Vacation	-n8 -q10 -u80 -r65536 -t4096

表 3: 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%

(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，既存モデル（B）の実行サイクル数を 1 として正規化している．なお，フルシステムシミュレータ上でマルチスレッドを用いた動作のシミュレーションを行うには，性能のばらつきを考慮する必要がある [23]．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

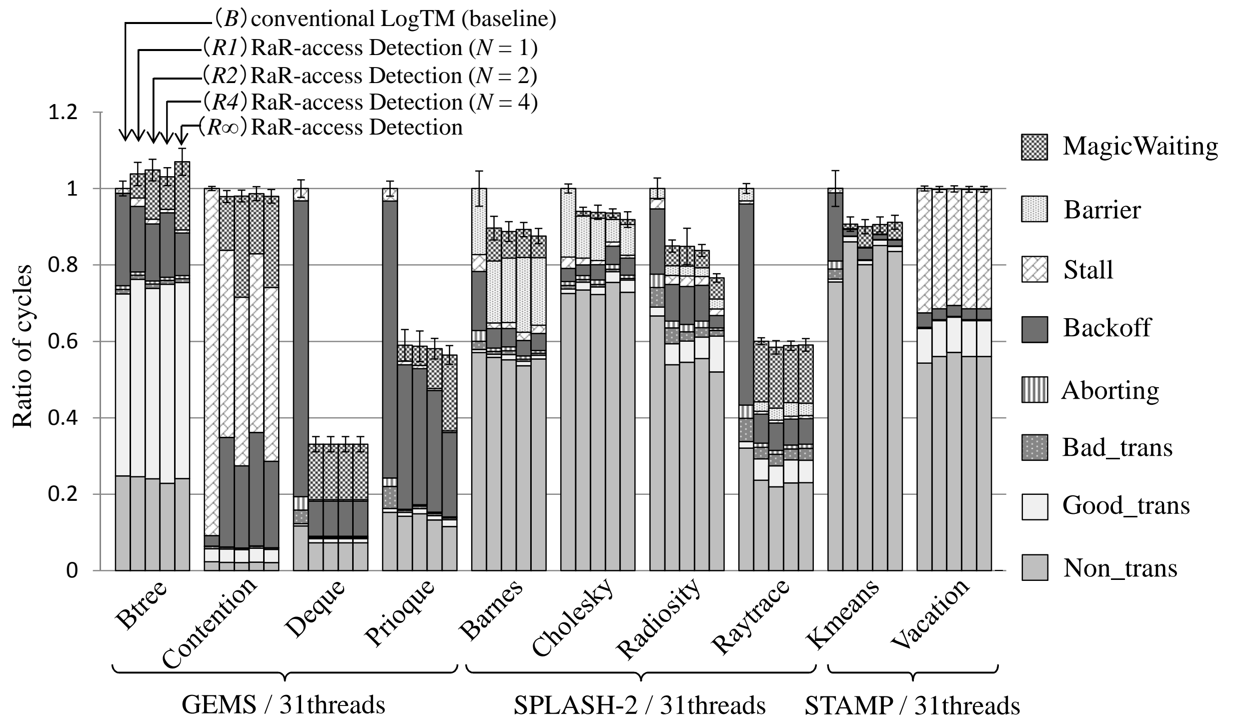


図 16: 各プログラムにおけるサイクル数比

求めた．信頼区間はグラフ中にエラーバーで示す．また，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，図 16 とは別に 6.4 節で詳細に述べる．

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詳細な検証を行う．

表 4: Contention における最大アボート繰り返し回数

Contention/31thr	(B)	(R_1)
最大アボート繰り返し回数	15	22

6.3　考察

GEMS microbench

まず GEMS microbench では，各提案モデルにおいて Deque，Prioqueue で既存モデル対し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ていたため，(R_1) のようにアドレスの登録数が少なくても Futile Stall と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低下した．この Btree には，2 種類のトランザクション（$Tx.X$，$Tx.Y$）が存在し，$Tx.X$ 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，$Tx.Y$ 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 $Tx.X$ もしくは $Tx.X$ と $Tx.Y$ 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 $Tx.Y$ 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いため，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既存モデルより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，コストが膨大となってしまうため，この性能低下に対処する必要性は低いと考えられる．

また，Contention では Stall サイクル数が大きく削減されたが，同時に Backoff サイクル数が増加し，既存モデル（B）に対する大きな性能向上は得られなかった．ここで，提案モデル（R_1）の Contention における最大アボート繰り返し回数を**表 4** に示す．この結果から，(R_1) では同一トランザクションがアボートを繰り返す回数が増加していることが分かる．これは，Contention に含まれている実行命令数の非常に多いトランザクションが原因であると考えられる．提案手法によってこのようなトランザクションが逐次実行の対象として選択されると，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する状況が多発する．このような場合，4.2.2項に示したように後者のスレッドが実行トランザクションをアボートすることになるが，当該トランザクションを再実行したとしても，前者のスレッドが実行トランザクションを終了していない可能性が高い．そのため，これらのスレッド間で再び競合が発生してアボートが繰り返されてしまうことから，(*B*) よりもアボート繰り返し回数が増加し，これが Backoff サイクル数の大幅な増大につながったと考えられる．したがって今後，トランザクション内の実行命令数によって逐次実行の対象を適切に選択する手法を考案していく必要がある．

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と考えられる．また，Cholesky では主に Barrier サイクル数が減少した．これは，本手法を適用して 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とるために他のスレッドを待ち続ける期間が少な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 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，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考えられる．

STAMP

STAMP ベンチマークでは，本手法によって Kmeans の実行サイクル数を削減することができた．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 自体は他のベンチマークと比較すると規模が小さいプログラムであるため，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

表 5: （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	-	-

少なかったと考えられる．一方，Vacation では本手法による性能向上が得られなかったが，このプログラム中には本論文に示したような，Write アクセスに先立って Read アクセスするトランザクション処理が含まれていなかったことから，検出すべき RaR アクセスが存在しなかったと考えられる．

6.4 RaR addr. の参照コスト

5.2 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考察する．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 5** に示す．5.2.3 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，アドレス登録時のビットシフトに要する時間を考慮したとしても，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 $T = 3$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，$1{,}257{,}086 \times 3 =$ 約 380 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ている．

7 おわりに

本論文では，既存のハードウェア·トランザクショナル·メモリを拡張して，Read/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ことで，あえて実行並列度を低下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．提案手法によって性能向上が得られた要因として，既存のハードウェア・トランザクショナル・メモリ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の発生を抑制したことによる，再実行サイクル数の大幅な削減が挙げられる．また，RaR アクセスを検出するために記憶するアドレス数を 1 とした場合でも性能向上は十分に得られていることから，提案手法を実現するためのハードウェアコストは少なく，実現性が高いことが分かった．

しかし提案手法を適用することで，本来であればトランザクションを並列実行すべき状況でも，それらのトランザクションを逐次的に実行してしまう場合があった．また，実行命令数の多いトランザクションを逐次実行の対象として選択すると，再開順序制御に関与していないスレッドが同一トランザクションを繰り返しアボートしてしまう場合も見受けられた．したがって今後，これらの原因をより明確にして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方法を考案していく．さらに，提案手法では再開順序を制御する際に遊休状態となるスレッドが存在していた．したがって，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．

謝辞

本研究を進めていくにあたって，いつも優しい眼差しで見守ってくださった松尾啓志教授に深く感謝致します．そして，研究の方向性や内容に関する的確な助言と，厳しさの中にも愛のある丁寧な御指導を賜わった津邑公暁准教授に深く感謝致します．検討会や日常生活において様々なアドバイスおよび多大なご尽力を頂いた齋藤彰一准教授，松井俊浩准教授，梶岡慎輔助教に深く感謝致します．また，私のような者を研究室の仲間として歓迎してくださった松尾·津邑研究室，齋藤研究室，松井研究室の方々にはどれだけ感謝してもしきれません．特に，江藤正通氏，堀場匠一朗氏と研究につ

いて語り合えたことは何ものにもかえがたい思い出です．肉体の鍛錬をともにしてくださった小田遼亮氏，水野航氏，神村和敬氏，大平真司氏に深く感謝致します．242 部屋の主としてその力をいかんなく発揮してくださった吉田健二氏，そして 242 部屋で生活を共にした柳田大輝氏，佐藤貫大氏，鈴木大輝氏，谷藤拓麻氏に深く感謝致します．常日頃から御世話になった山田龍寛氏，祖父江宏祐氏，里見優樹氏，澤田晃平氏，内山寛章氏に深く感謝致します．研究が辛くとも，共に切磋琢磨して壁を乗り越えてきた B4 の皆様に深く感謝致します．最後となりますが，私の研究生活に関わっていただいた全ての方々に深く感謝致します．

著者発表論文

報文

1. 橋本 高志良，江藤 正通，鈴木 大輝，堀場 匠一朗，津邑 公暁，松尾 啓志：“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”，情処研報，Vol.2013-ARC-204,（to appear）（Mar. 2013）
2. 鈴木 大輝，橋本 高志良，堀場 匠一朗，江藤 正通，津邑 公暁，松尾 啓志：“ハードウェアトランザクショナルメモリにおける競合パターンに応じた競合再発抑制手法の適用”，情処研報，Vol.2013-ARC-204,（to appear）（Mar. 2013）

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] International Business Machines Corporation: *IBM System BlueGene Solution BlueGene/Q Application Development*, 1 edition (2012).

[3] Intel Corporation: *Intel Architecture Instruction Set Extensions Programming Reference, Chapter 8: Transactional Synchronization Extensions.* (2012).

[4] Shavit, N. and Touitou, D.: Software Transactional Memory, *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pp. 204–213 (1995).

[5] Sweazey, P. and Smith, A. J.: A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus, *Proc. 13th Annual Int'l. Symp. on Computer Architecture (ISCA'86)*, pp. 414–423 (1986).

[6] Censier, L. M. and Feautrier, P.: A New Solution to Coherence Problems in Multicache Systems, *IEEE Trans. on Computers*, Vol. C-27, No. 12, pp. 1112–1118 (1978).

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

[8] 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).

[9] McDonald, A., Chung, J., Caristrom, B. D., Minh, C. C., Chafi, H., Kozyrakis, C. and Olukotun., K.: Architectural Semantics for Practical Transactional Memory, *ISCA '06 Proceedings of the 33rd annual international symposium on Computer Architecture*, pp. 53–65 (2006).

[10] Moss, E. and Hosking., T.: Nested Transactional Memory: Model and Preliminary Architecture Sketches., *In OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Langauges*, pp. 186–201 (2005).

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

[12] Tomic, S., Perfumo, C., Kulkami, C., Armejach, A., Cristal, A., Unsal, O., Harris, T. and Valero., M.: Eazyhtm, Eager-lazy Hardware Transactional Memory, *In Procs. of the 42nd Symp on Microarchitecture (MICRO'09)*, pp. 145–155 (2009).

[13] 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).

[14] 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).

[15] 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).

[16] 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).

[17] 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).

[18] 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).

[19] 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).

[20] 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).

[21] 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).

[22] Gramoli.V and Guerraoui.R: Transactions, http://lpdserver.epfl.ch/transactions/wiki/doku.php?id=stamp (2011).

[23] 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).