Vol.2012-ARC-201 No.1
2012/8/1

競合検出単位の細粒度化による トランザクショナルメモリの高速化

堀場 匠一朗[1]　浅井 宏樹[1,†1]　江藤 正通[1]　津邑 公暁[1]　松尾 啓志[1]

概要：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制御にロックが広く利用されてきたが，その場合，デッドロックの発生や並列性の低下等の問題がある．そこで，ロックを用いない並行性制御機構としてトランザクショナル・メモリ（TM） が提案されている．これをハードウェアで実現する HTM では，トランザクション内で発生したリードおよびライトアクセスの情報をキャッシュライン単位で記憶するため，他のトランザクションでアクセスされた変数とは異なる変数にアクセスしたとしても，それらの変数が同一キャッシュライン上に配置されていた場合，競合として判定されてしまう．そこで本稿では，競合の検出単位を細粒度化し，本来競合ではないアクセスが競合として検出されてしまうことを防止する手法を提案する．評価の結果，既存手法に比べて，最大 89.4%，16 スレッドで平均 26.3%の実行サイクル数の削減を確認した．

1. はじめに

これまでのプロセッサ高速化技術は，スーパスカラに代表されるような (Instruction Level Parallelism: ILP) に基づくさまざまな手法を中心としつつ，集積回路の微細化による高クロック化を半導体技術の向上により実現することで支えられてきた．しかしながらプログラム中の ILP には限界があり，また消費電力や配線遅延の相対的増大により，クロック向上も頭打ちになりつつある．この流れを受け，単一チップ上に複数のプロセッサ・コアを集積したマルチコア・プロセッサが広く普及してきている．

このようなマルチコア環境における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制御する必要があり，そのための機構として一般的にロックが用いられている．しかし，ロックを用いたプログラミングでは，デッドロックが発生したり，最適なロックの粒度を設定しなければ並列性が低下するといった問題がある．このため，ロックはプログラマにとって必ずしも利用しやすい機構ではない．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 [1] が提案されている．

このトランザクショナル・メモリをハードウェアで実現する HTM（Hardware Transactionl Memory）では一般的に，各キャッシュラインに対して read および write ビットという，トランザクション内で発生したリードおよびライトアクセスの有無を記憶するフィールドが追加されている．そして，キャッシュリクエストを受け取ったときにこれらのビットを参照することで，競合検出を実現する．しかし，キャッシュライン単位で競合を検査するため, 他のトランザクション内でアクセスされた変数とは異なる変数にアクセスしたとしても，それらの変数が同一キャッシュライン上に配置されているならば，競合として判定されてしまう．

したがって，このように誤って検出された偽の競合においても，これを検出したスレッドは，競合相手のトランザクションが終了するまで，自身が実行するトランザクションを一時的に停止してしまうという問題がある．さらに，プログラマにより並行性制御の対象として指定されなかった変数に対するアクセスでも競合として判定される可能性があり，そのような変数はトランザクション内部でアクセスされることが保証されない．このため，トランザクション外の処理にも影響を与えてしまう可能性がある．

そこで，本稿では既存のハードウェアを拡張することで競合を検出する単位を細粒度化する手法を提案する．これにより，本来競合ではないアクセスが競合として検出されてしまうことを防止する．

2. トランザクショナル・メモリの概要

本章では，本研究の対象となるトランザクショナル・メモリと，それをハードウェアで実現する HTM の概念について説明する．

2.1 トランザクショナル・メモリ

マルチコア・プロセッサにおける共有メモリ型並列プロ

[1] 名古屋工業大学
Nagoya Institute of Technology
[†1] 現在，株式会社デンソー
Presently with DENSO CORPORATION

グラミングでは，共有リソースへのアクセス制御にロックが用いられてきたが，これにはデッドロックの発生や並列性の低下等の問題がある．

そこで，ロックを用いない並行性制御機構である**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り，クリティカルセクションを含む一連の命令列を投機的に実行する．この命令列は，以下の 2 つの性質を満たす**トランザクション**として定義される．

シリアライザビリティ（直列可能性）： 並行実行されたトランザクションの実行結果は，当該トランザクションを直列に実行した場合と同じであり，全てのスレッドにおいて同一の順序で観測される．

アトミシティ（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，もしくは全く実行されないかのいずれかでなければならず，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 は各トランザクション内でアクセスされるメモリアドレスを監視する．ここで，複数のトランザクション内において同一アドレスへのアクセスが確認されると，これがトランザクションの性質を満足しない場合，**競合**として判定される．この操作を**競合検出**という．競合を検出した場合は，片方のトランザクションの実行を一時的に停止する．これを**ストール**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，デッドロックの可能性があると判断された場合，片方のトランザクションの実行結果を全て破棄する．これを**アボート**という．そして，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再実行する．一方でトランザクションが終了するまでに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すべての結果をメモリに反映させる．これを**コミット**という．

TM はこ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に実行することができる．なお，TM で行われる競合の検出，コミット，およびアボート等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ることで実現される．これらのうち，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，ソフトウェア上に実装された TM はソフトウェア・トランザクショナル・メモリ（STM）[2] と呼ばれる．STM では，HTM のような特別なハードウェア拡張は必要ないが，TM 上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，オーバヘッドが増大する．したがって，HTM は STM に比べて速度性能が高い．

2.2 競合の検出と解決

競合を検出するためには，どのアドレスがトランザクション内でアクセスされたかをトランザクション毎に記憶する必要がある．そのため，HTM では一般的に，各キャッシュラインに対して **read ビット**および **write ビット**と呼ばれるフィールドが追加されている．各ビットはトランザクション内で当該キャッシュラインに対するリードアクセスおよびライトアクセスが発生した場合にそれぞれセットされ，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に，HTM ではキャッシュの一貫性を保持するプロトコルを拡張している．一貫性プロトコルでは，スレッドがあるメモリアドレスにアクセスする場合，キャッシュラインの状態を変更させるリクエストが他の各スレッドに送信される．このとき，各スレッドはリクエストを受信すると，キャッシュラインの状態を変更する前に，キャッシュに追加された read および write ビットを参照する．これにより，他のトランザクションとの競合を監視する．なお，以下の 3 パターンのアクセスが発生した場合を競合として判定する．

Read after Write: write ビットがセットされているアドレスに対するリードアクセス．

Write after Read: read ビットがセットされているアドレスに対するライトアクセス．

Write after Write: write ビットがセットされているアドレスに対するライトアクセス．

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．NACK を受信したスレッドは自身のアクセスで競合が発生したことを知るが，すぐにはアボートせず，トランザクションをストールする．このとき，トランザクションをストール中のスレッドは競合したアドレスに対するリクエストを送信しつづけることで，相手のトランザクションが終了したかどうかを監視する．一方で，競合が検出されなかった場合は従来の一貫性プロトコルに従う．例えば，無効化リクエストに対しては ACK が返信され，共有リクエストに対しては共有されるデータが返信される．なお，この競合検出方式は，そ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: トランザクションが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．

実際に競合が発生してからそれを検出するまでの時間が長くなる lazy 方式では，無駄な実行が増大してしまい効率が悪い．

2.3 データのバージョン管理

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

と更新前の値とを併存させる必要がある．そこで HTM では，トランザクション内で発生したライト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の領域に保持する．このようなデータの管理はバージョン管理と呼ばれ，以下の 2 つの方式に大別される．

Eager Version Management: 書き換え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，アボート時にはバックアップされた値をメモリにリストアする必要がある．

Lazy Version Management: 書き換え前の古い値をメモリに残し，新しい値を別領域に登録する．アボートは高速に行えるが，コミット時にメモリへの値のコピーが必要となる．

ここで，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コストを払う考え方である．アボートが繰り返し発生してしまうようなプログラムでは不利となる場合もあるが，lazy 方式ではコミットのためのオーバヘッドは削減の余地がほぼないのに対し，eager 方式では競合やアボートの発生自体を抑制することで性能向上できる余地が大きいと考えられる．

よって本稿では，競合検出方式とバージョン管理方式について，eager 方式同士を組み合わせた（eager/eager）HTM の実装の一つである **Log-based Transactional Memory（LogTM）** [3] に対し，提案する手法を実装し，評価する．ただし，本提案手法はこれら 2 方式のいずれの組み合わせに対しても適用可能である。

3. 関連研究

マルチコア環境において，複数のスレッドが異なるデータにアクセスする際，それらのデータが同一キャッシュライン上に存在していた場合，false sharing[4] とよばれる偽の共有状態が引き起こされることが知られている．この問題に対する一般的な解決策として，パディングにより異なるデータの間にダミーのデータを挿入し，それらを別々のキャッシュラインに配置するという手法 [5] が存在する．しかし，この手法はメモリ使用量を増大させ，また，ある領域の近傍データがキャッシュ上の別のラインに配置されることで，空間的局所性を低下させる可能性がある．

また，複数のトランザクションが並行に投機実行される状況下においては，この false sharing によって，本来は存在しない競合が検出される．これにより，競合を検出したスレッドは実行中のトランザクションを停止しなければならないため，その実行が直列化されてしまうという問題がある．

このような競合の発生を解決するために，異なるデータが同一キャッシュライン上に存在していた場合でも，それらが自身以外のトランザクションによってまだ更新されていないものと予測し，それらの値を使用してプログラムの実行を継続する手法 [6] が提案されている．これにより，false sharing に起因する，競合の誤検出によるストールを排除することができる．また，予測値として使用するデータをビットを用いて監視し，コミット時にこの値の更新が検出された場合には，トランザクションをアボートすることでアトミシティを維持する．

しかし，このような値予測により並列性を向上させる場合，他のトランザクションのストア命令により，一度でも予測ミスが発生した場合には，トランザクションを必ずアボートする必要がある．したがって，値予測の失敗が，バックアップされた値をメモリにリストアする操作等によるオーバヘッドを引き起こすと考えられる．また，トランザクションの実行途中に競合が検出されアボートが発生する場合，ある同一キャッシュライン上に存在する異なるデータの一部がトランザクション内で変更されていたならば，トランザクション開始前の値を，変更のあった値に限定して復元する必要がある．しかし，[6] の手法はトランザクションをコミットする前に競合が検出される状況を想定していないため，eager な競合検出方式への適用が不可能である．

一方，本稿で提案する手法は，競合検出単位を細粒度とすることで，実際に同一アドレスに対する競合が発生しているかどうかをより厳密に検査する．これにより，本来競合ではないアクセスを競合として検出されることを防止する．また，アボート時における部分的な値の復元を実現することで，eager な競合検出方式への適用も可能である点が従来手法とは異なる．

4. 競合検出単位の細粒度化

本章では，一般的な HTM が持つ問題点を指摘し，それを解決するために，競合検出単位を細粒度化する手法を提案する．

4.1 既存の HTM が持つ問題点

既存の HTM ではキャッシュライン単位で競合を検査するため，複数のスレッドが異なるデータにアクセスしたとしても，それらが同一キャッシュライン上に存在していた場合は競合として判定されてしまうという問題がある．この問題を，eager/eager 方式の LogTM において，2 つのスレッドが異なるデータにアクセスする場合を例に説明する．**図 1** 中の Core と Cache はそれぞれプロセッサ・コアと各コアが持つローカルキャッシュを簡略化して示したものであり，Shared Memory は 2 つのコアに共有されている主記憶を示している．また，Core1 には Thread1 が，Core2 には Thread2 がそれぞれ割り当てられており，各スレッドでは**図 2** に示すようなプログラムが実行されているとする．なお，図 2 中の `BEGIN_XACT;` および `COMMIT_XACT;` はそれぞれトランザクションの開始および終了を表しており，トランザクションの範囲を定義している．また，図 2 中の Time はプログラム内の各式が実行される時刻

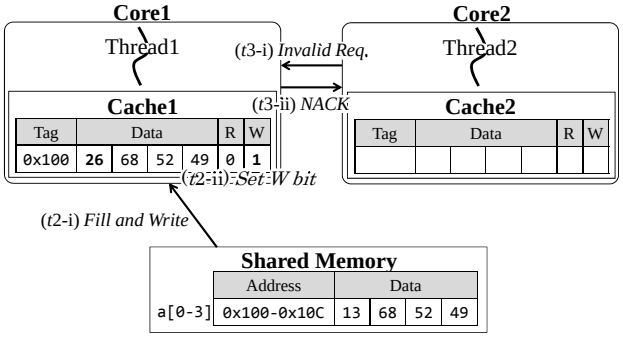

図 1 異なる変数へのアクセスによる競合

Time	Thread1	Thread2
$t1$	`BEGIN_XACT;`	`BEGIN_XACT;`
$t2$	`a[0]=26;`	`...`
$t3$	`...`	`a[2]=70;`
$t4$	`...`	`COMMIT_XACT;`

図 2 各スレッドの実行プログラム

($t1 < t2 < t3 < t4$) を表している．

まず，2 つのスレッドがトランザクションの実行を開始し（時刻 $t1$），Thread1 が `a[0]` へ値を代入する（$t2$）．このとき，Cache1 には `a[0]` が保持されていないため， 主記憶の 0x100 番地から始まるデータがキャッシュされ，`a[0]` の値が更新される（図 1 中 $t2$-i）．LogTM ではアドレス 0x100 番地，および更新される前のキャッシュラインのデータが**ログ**と呼ばれる領域に退避されるが，図中では省略する．また，トランザクション内でライトアクセスが発生したため，0x100 番地に対応するキャッシュラインの write ビットがセットされる（$t2$-ii）．

次に，Thread2 が `a[0]` と同一キャッシュライン上に存在する `a[2]` へ値の代入を試みる（$t3$）．このとき，一貫性プロトコルに従って 0x100 番地のラインに対する無効化リクエストが Thread2 から送信される（$t3$-i）リクエストを受け取った Thread1 は 0x100 番地に対応するラインの read および write ビットを参照する．このとき，write ビットがセットされているため競合として判定され，Thread2 に対して NACK が返信される（$t3$-ii）．

このように，複数のスレッドがそれぞれ異なるアドレスに対応する変数にアクセスしたとしても，それらの変数が同一キャッシュライン上に存在していた場合，競合として判定されてしまう．

また，スレッドローカルに宣言された異なる変数は通常，同一キャッシュライン上に配置されることはない．しかし，それらの変数のためのメモリ領域が malloc() により一次元のヒープ領域から確保される場合，スレッドローカルに定義された異なる変数や，スレッドローカルな変数とグローバルな変数とが，同一キャッシュライン上に存在する可能性がある．

ここで，LogTM では，NACK を受信したすべてのスレッドはストールすることで一時的に実行を停止する必要があるため，これが性能の悪化を引き起こす可能性がある．本稿では，本来競合ではないアクセスにより発生するストールを **false stall** と呼ぶ．なお，この false stall はトランザクションの範囲内だけでなくトランザクションの範囲外でも発生する可能性がある．

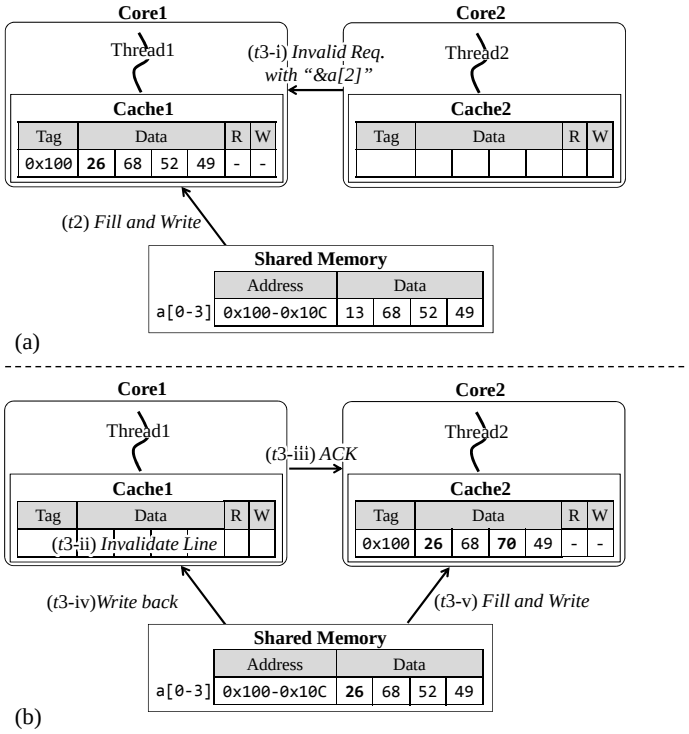

図 3 同一キャッシュライン上の変数アクセスを許可

そこで，既存のハードウェアを拡張し，メモリ上に配置されたデータに対して細粒度で競合を検査する手法を提案する．これにより，異なる変数に対するメモリアクセスを判別し，本来競合ではないアクセスが競合として検出されてしまうことを防止する．

4.2 競合検査の動作モデル

提案する競合の検査手法について，図 2 に示すトランザクションを含むプログラムを， **図 3**（a）に示す 2 つのスレッドがそれぞれ実行する例を用いて説明する．このプログラムが実行されると，まず 2 つのスレッドはトランザクションを開始し（$t1$），続いて Thread1 が `a[0]` へ値を代入する（$t2$）．すると，0x100 番地のデータがキャッシュされ，`a[0]` の値が更新される（図 3 中（a）$t2$）．

次に，Thread2 が `a[2]` へ値の代入を試みる（$t3$）．すると，Thread1 が持つ 0x100 番地のキャッシュラインを無効化するリクエストが送信される（$t3$-i）．このとき，既存手法では，リクエストには無効化するキャッシュラインラインアドレス 0x100 の情報が含まれているが，アクセスする `a[2]` のアドレス情報は含まれていない．そのため，Thread1 は `a[2]` に対するアクセスが，本質的に競合であるかどうかを検査することができない．そこで，提案手法

ではリクエストを送信すると同時に，a[2] のアドレスも Thread1 へ送信する．さらに，Thread1 は予め，トランザクション内で発生したリードおよびライトアクセスの有無に関する情報を保持しておき，リクエストと a[2] のアドレスを受け取った際にこれを参照することで，a[2] との競合を検査する．

この例では，Thread1 は a[2] に対しリードおよびライトアクセスしていないため，Thread2 による a[2] へのアクセスとは競合しないと判定される．したがって，Thread1 は既存の一貫性プロトコルに従って，0x100 番地のキャッシュラインを無効化し（図 3（b）$t3$-ii），Thread2 に対して ACK を送信し（$t3$-iii），キャッシュラインをライトバックする（$t3$-iv）．その後，Thread2 は 0x100 番地のラインをキャッシュし，a[2] の値を更新する（$t3$-v）．

4.3 アボート時の操作

LogTM ではトランザクションをアボートさせる場合，ログに退避された値を元のメモリアドレスへ書き戻すことで，トランザクション内で更新した値を更新前の状態に復元する．このとき，ログ領域に退避されたキャッシュラインのデータが全て復元されるため，他のスレッドで更新された同一ライン上の変数の値も，ログに退避された値で上書きされてしまう．これは，既存の競合検出方法ではキャッシュライン単位で他のスレッドからのアクセスを制御しており，トランザクション中でライトアクセスされたラインが他のスレッドにアクセスされることを考慮していないためである．そこで，提案手法では自身のトランザクション内で更新した値のみを書き戻すことで，キャッシュの一貫性を保持する．

5. 実装

本章では 4 章で提案した競合検出と，アボート時の操作を実現するために必要なハードウェア，およびそれらの動作について説明する．

5.1 ハードウェア拡張

本提案手法では，細粒度で競合を検査する必要があるキャッシュラインに限定して，リードおよびライトアクセス情報を記憶するための小容量のハードウェアを各コアに追加する．これを **R/W テーブル**と呼ぶ．また，キャッシュラインから R/W テーブルへ一意にアクセスできるように，R/W テーブルのインデクスを保持する **Ptr** フィールドをキャッシュの各ラインに追加する．

これらの機構を追加したハードウェア構成を**図 4** に示す．R/W テーブルでは，既存手法で競合検査の対象であったキャッシュラインというメモリブロックを，提案手法では N 個に分割する．そして，分割後の各要素を競合検査の最小単位として扱うことで，ひとつのキャッシュライン上に存在する N 個のチャンクに対する競合を検査することができる．この R/W テーブルは，キャッシュタグを記憶

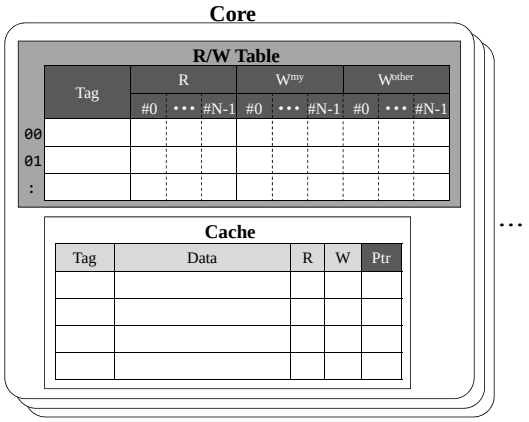

図 4 提案するアーキテクチャ構成

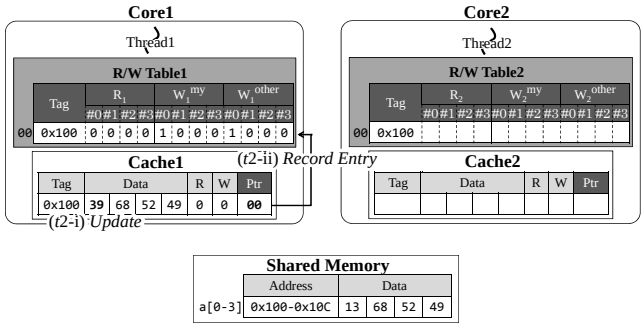

図 5 R/W テーブルのエントリを登録

する Tag，リードアクセス情報を記憶する R，自コアに割り当てられたスレッドが実行するトランザクション内のライトアクセス情報を記憶する W^{my}，他スレッドで実行されるトランザクション内で発生したライトアクセス情報を記憶する W^{other} フィールドを持つ．R，W^{my} および W^{other} は，それぞれのアクセス情報を保持するための N ビット（#0〜#$N-1$）のレジスタにより構成される．

5.2 R/W テーブルのエントリ登録

提案手法では競合が発生したラインのみを R/W テーブルで管理する．しかし，競合が発生するキャッシュラインを判定するためには，静的にプログラムを解析し，競合する可能性のある変数を事前に検出する必要がある．

そこで提案手法では，過去に一度でも競合が発生したキャッシュラインを R/W テーブルによる管理対象とする．このため，あるキャッシュラインで最初に発生した競合に限り，そのアクセスを既存手法と同様にライン単位で検出する必要があるが，静的解析のためのコストは不要となる．

なお，1 度目の競合時には，false stall が発生する可能性があるが，以降では同じキャッシュラインにおいて細粒度で競合を検査することができるため，性能に大きな影響を与えないと考えられる．このとき，2 度目以降の競合でそのラインを R/W テーブルにより管理するために，1 度目の競合の時点で当該ラインの Ptr に R/W テーブルのインデクスがセットされる．

ではここで，2 つのスレッドが異なるトランザクションを実行する例を**図 5** に示す．図 5 中の Thread1 および Thread2 はそれぞれ 4.1 節 図 2 のトランザクションを実行するとする．なお，分割数 $N = 4$ とし，16Bytes のキャッシュライン上に int 型のデータが 4 つ保持される場合を考える．

まず，2 つのスレッドでトランザクションの実行が開始される（$t1$）．その後に Thread1 上で `a[0]` への代入が実行され（$t2$），`a[0]` の値が更新される（$t2$-i）．このとき，0x100 番地のキャッシュラインの Ptr がセットされているとすると，このラインに対する競合が過去に発生しており，これ以降 R/W テーブルにより競合を検査する必要があることがわかる．そこで，R/W テーブルのインデクス 00 に対してエントリを登録する（$t2$-ii）．登録するエントリにはラインの持つ Tag がセットされる．そして，変数に対応した R_0，W_0^{my} フィールドをセットする．この例では，Thread1 は `a[0]` にライトアクセスするため，$W_1^{my}[\#0]$ にのみ 1 がセットされ，$W_1^{my}[\#1]$〜$W_1^{my}[\#3]$ には 0 がセットされる．また，この時点ではまだ 0x100 番地のキャッシュラインは他のトランザクションからアクセスされていないため，W_1^{other} フィールドには 0 がセットされる．

5.3 R/W テーブルのエントリ参照による競合の検査

提案手法では，リクエストを受け取ったときに，そのラインに対応する R/W テーブルエントリが存在していたならば，細粒度で競合を検査する．ここで, 前述した図 2 のトランザクションを実行する例を再び考える．現在，$t2$ まで進行しているとし，続いて，Thread2 が `a[2]` に対して値の代入を試みるとする（$t3$）．このとき，**図 6**（a）で示されるように，`a[2]` のアドレスを付加した無効化リクエストが Thread2 から送信される（$t3$-i）．ここで，リクエストを受け取った Thread1 の R/W テーブルには，0x100 番地のラインに対応するエントリが存在しているため，そのエントリを参照することで競合を検査する（$t3$-ii）．このとき，`a[2]` に対応している $R_1[\#2]$，$W_1^{my}[\#2]$ および $W_1^{other}[\#2]$ には 0 がセットされているため，競合は検出されない．したがって，Thread1 は 0x100 番地を無効化し（図 6（b）（$t3$-iii））, 同時に 0x100 番地の Ptr もクリアする．しかし，R/W テーブルのエントリはクリアされずに維持される．したがって，もし今後 Cache1 が再び 0x100 番地のラインを保持したならば，R/W テーブル は 0x100 番地の Tag を記憶しているため，該当する R/W テーブルのインデクスを Ptr にセットし，以降の競合時に当該エントリを参照することができる．

さて, 競合の検査はリクエストを受信したスレッドが行うが，例では 0x100 番地のキャッシュラインが無効化されたため，今後 0x100 番地に対するリクエストが Thread1 に送信されることがなくなる．したがって，Thread1 は 0x100 番地上に存在する変数に対する競合を検査することができなくなる．そこで，Thread2 は Thread1 の実行するトランザクション内におけるリードおよびライトアクセス情報を保持することで，Thread1 における競合検査の責任を代わりに請け負う．そのため，Thread1 は Thread2 に返信する ACK に対して，0x100 番地のラインに対応する R/W テーブルの R_1 フィールドの値，および W_1^{my} と W_1^{other} フィールドの値の論理和を求めた値を付加する（$t3$-iv）．以降，W_1^{my} と W_1^{other} フィールドの論理和を W_1 と呼ぶ．この ACK が返信される際，Thread1 は 0x100 番地のラインを書き戻す（$t3$-v）．

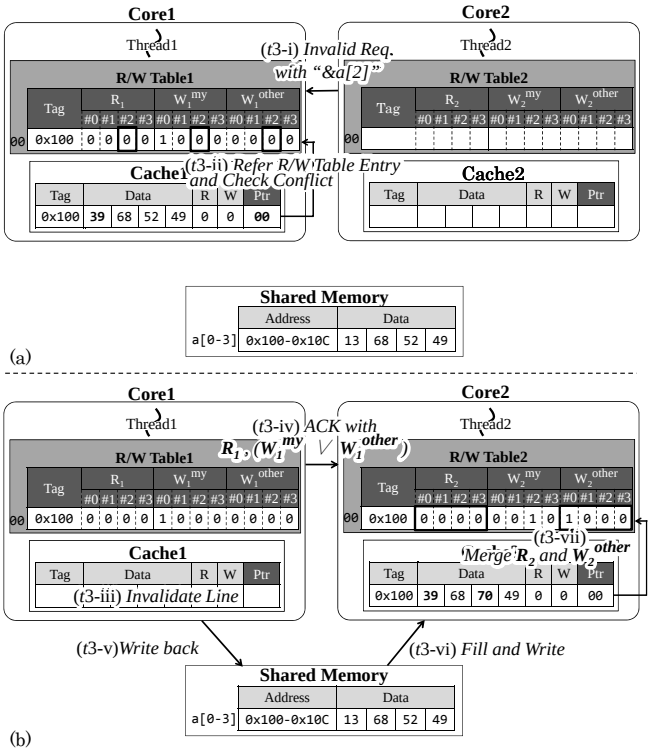

図 6 R/W テーブルを利用した競合の検査

この後，Thread2 は ACK の受信によりそのラインにアクセスできるようになったことを知るため，そのラインをキャッシュする（$t3$-vi）．さらに，Thread2 は ACK に付加された R_1 および W_1 の値により，0x100 番地が，過去に競合が発生したラインであることを知る．そのため，0x100 番地に対する R/W テーブルエントリを登録し，そのインデクスを Ptr に保持し，`a[2]` に対するライトアクセス情報を $W_2^{my}[\#2]$ に記憶する．ここで，Thread1 のトランザクションで過去に行われたリードおよびライトアクセス情報を保持するために，Thread2 は自身が保持する R_2 の値と受信した R_1 の値の論理和および W_2^{other} と W_1 の値の論理和を求め，それぞれを R_2 と W_2^{other} に上書きする（$t3$-vii）．これにより，以降，他のトランザクション内で発生した `a[0]` へのアクセスに対する競合を Thread2 が検査することができる．

5.4 R/W テーブルのエントリ破棄

R/W テーブルに記憶されるリードおよびライトアクセス情報は，実行中のトランザクション内で発生した固有の

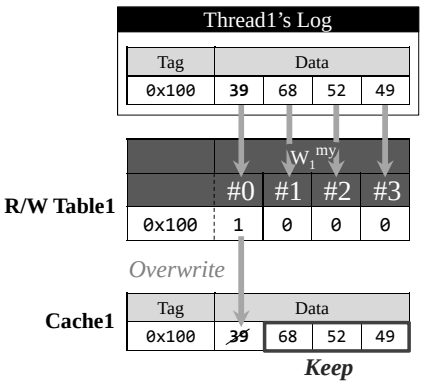

図 7 W^{my} による書き戻し対象の限定

情報である．そのため，トランザクションがコミットおよびアボートされた場合は R/W テーブル のエントリは全て破棄される．一方で，Ptr フィールドはクリアされずに維持される．

また，4.3 節で述べたように，アボート時における書き戻し対象はアボートされるトランザクション内で更新した変数のみとする必要がある．これを実現するために，ログに退避された更新前のキャッシュライン状態に対して，そのラインに対応する R/W テーブルエントリの W^{my} をマスキングすることで，書き戻す必要のある値を判別する．

例えば，5.3 節で説明した図 6 の例における Thread1 が，図 2 の $t4$ 以降にアボートされたとする．このとき，アボートにより書き戻すラインのデータに対して，W_1^{my} フィールドの値により**図 7** のようにマスクすることで，Thread1 のトランザクション内で更新された `a[0]` の値のみを復元することができる．

6. 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，それを実現するためのハードウェアコストおよびオーバヘッドを概算する．

6.1 評価環境

これまで述べた拡張を HTM の一実装である LogTM に実装し，シミュレーションによる評価を行った．評価には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[7] 3.0.31 と GEMS[8] 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パラメタを示す．

評価対象のプログラムは GEMS 付属の microbench に含まれる Prioque，Sortedlist に加えて，STAMP[9] ベンチマークプログラムから Kmeans，Vacation を用いた．各プログラムの入力を**表 2** に示す．

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
L1 I&D cache	32 KBytes
ways	4 ways
latency	3 cycle
line size	64 Bytes
L2 cache	8 MBytes
ways	8 ways
latency	34 cycles
line size	64 Bytes
L2 Directory	Full-bit vector sharers list
latency	6 cycles
Memory	4 GBytes
latency	500 cycles
Interconnect network	2D mesh topology
link latency	3 cycles
link bandwidth	64 Bytes

表 2 ベンチマークプログラムとその入力

Vacation	64K entries, 4K tasks, 8 queries, 10 rel, 80 users
Kmeans	40/40 clusters, thres. 0.05, 1000 12-dim points
Prioque	8192 ops
Sortedlist	500 ops, 64 length

表 3 各モデルにおけるサイクル削減率

thrs	(T_2)	(T_4)	(T_8)	(T_{16})
4	16.7%	17.5%	17.4%	17.1%
8	16.9%	16.2%	15.2%	15.2%
16	25.5%	25.7%	25.1%	26.3%
31	47.8%	48.6%	48.4%	48.6%

6.2 評価結果

評価の結果を**表 3** および**図 8** に示す．図 8 のグラフの凡例はサイクル数の内訳を示しており，FALSE STALL-XACT はトランザクションの範囲内で発生した false stall，FALSE STALL-NONXACT はトランザクションの範囲外で発生した false stall， XACT は FALSE STALL-XACT 以外のトランザクションの範囲内の実行に要したサイクル数，NONXACT は FALSE STALL-NONXACT 以外のトランザクションの範囲外の実行に要したサイクル数をそれぞれ示している．

各ベンチマークプログラムを 4，8，16，31 スレッドで実行した結果ごとにまとめて示しており，各ベンチマークプログラムとスレッド数との組み合わせによる結果をそれぞれ 5 本のグラフで表している．ただし，STAMP は 2 の冪乗数のスレッドでしか動作しないため，STAMP に限り 4，8，16 スレッドで評価した．5 本のグラフはそれぞれ左から順に

(B) 既存モデル（ベースライン）
(T_2) 提案モデル：キャッシュライン 2 分割（$N = 2$）

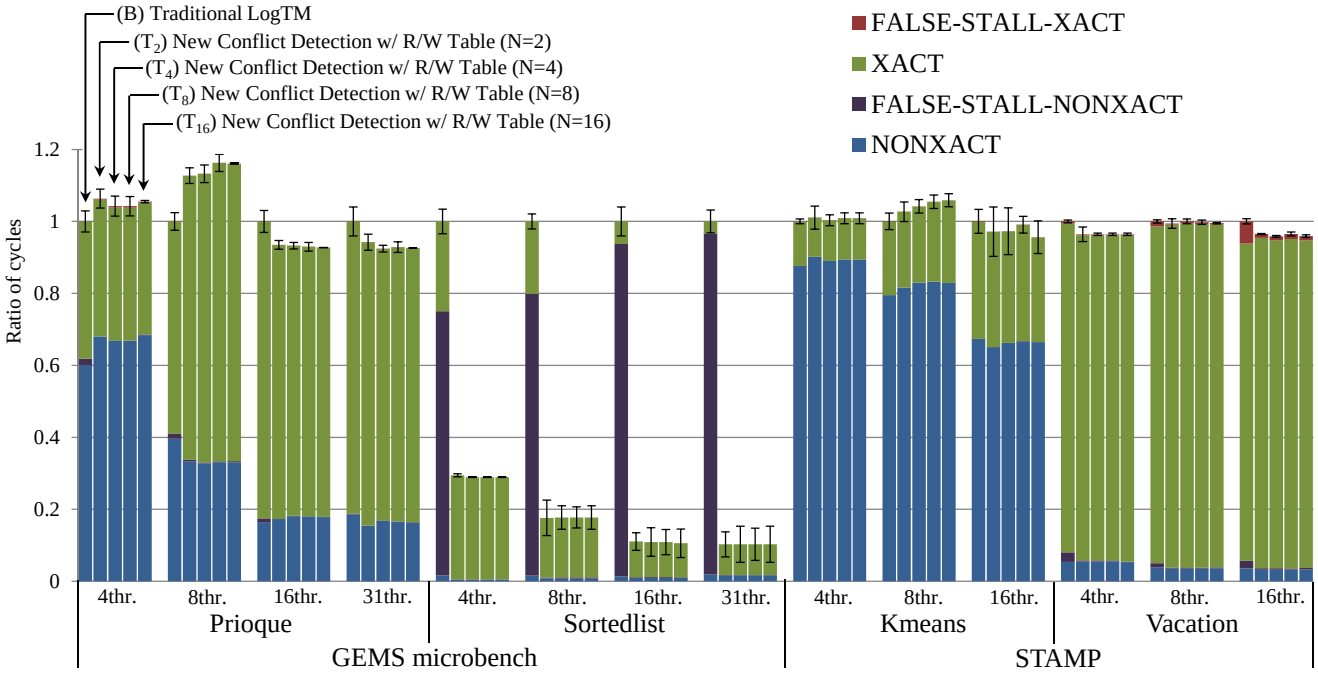

図 8 評価結果

($\mathbf{T}_4$) 提案モデル：キャッシュライン 4 分割（$N = 4$）
($\mathbf{T}_8$) 提案モデル：キャッシュライン 8 分割（$N = 8$）
($\mathbf{T}_{16}$) 提案モデル：キャッシュライン 16 分割（$N = 16$）
の実行に要した総サイクル数を表し，各サイクル数は (B) の実行サイクル数を 1 として正規化した．提案手法については，キャッシュラインを 2，4，8，16 個に分割して競合検出単位としたものを，それぞれ評価した．

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場合は性能のばらつきを考慮しなければならない [10]．したがって，各評価対象につき試行を 10 回繰り返し，得られたサイクル数から平均値と 95%の信頼区間を求めた．平均値をグラフの縦軸に，信頼区間をグラフ中のエラーバーで示している．

評価の結果，Prioque および Kmeans では特にスレッド数が少ない場合において総実行サイクル数が悪化する場合が見られたものの，これらの 4 つのベンチマークに対し FALSE STALL-NONXACT の削減を確認した．結果として，全プログラムを提案モデル (T_{16}) において 16 スレッドで実行した場合，平均で 26.3%，89.4%の実行サイクル数が削減された．

6.3 考察

以下，各ベンチマーク別に詳細な検証を行う．

GEMS microbench

まず，FALSE STALL-NONXACT について注目すると，Prioque，Sortedlist 両方のベンチマークにおいて，そのサイクル数が削減されていることがわかる．特に，FALSE STALL-NONXACT の占める割合が非常に大きい Sortedlist では，提案手法により大幅にサイクル数が削減された．

また，Sortedlist では提案モデル (T_2) でも，より細かいキャッシュライン分割数の場合とほぼ同量の FALSE STALL-NONXACT が削減され，総実行サイクル数もこれらの提案モデルと同等の結果を得ることができた．この原因を調査したところ，ライブラリ関数 srandom() 内の変数のために確保された領域と，リスト構造のために確保された領域が同一キャッシュライン上に存在しており，前者の末尾部分がキャッシュラインの前半，後者の先頭部分が同ラインの後半の領域にそれぞれ配置されていたことが分かった．これより，少なくとも今回評価したプログラムに関しては，提案モデル (T_{16}) ほどの細かい単位で競合を検査する必要がないと考えられる．

一方で，Prioque を 8 スレッドで実行した場合には，XACT が増大する傾向が見られた．この原因を調査した結果，既存モデル (B) に比べてストールサイクル数が増大していることが分かった．これは，本提案手法により FALSE STALL-NONXACT の発生を防止したことで，トランザクション外でスレッドが停止することがなくなり，トランザクション実行の並列度が増加したことで，却って競合が発生しやすい状態が生まれたためであると考えられる．

また，Prioque では 4 スレッドで実行した場合に NONXACT の増加が見られた．これは，トランザクション外を実行中のスレッドが L1 キャッシュミスを頻発させていたことが原因であった．共有状態にあるキャッシュラインへの書き込みが発生した場合，キャッシュ・コヒーレンス・プロトコルに従い，そのキャッシュラインを共有しているコアは当該ラインに対する無効化要求を受信する．ここで，提案手法により競合検出単位を細粒度化することで，異なるデータが同一ライン上に存在する場合でも，そのようなデータへのメモリアクセスが成功するようになる．この場合，以前そのラインを共有していたコアの実行するス

表 4 (T_{16}) における R/W テーブルの使用エントリ数

GEMS	/thrs	最大	平均	STAMP	/thrs	最大	平均
Prioque	/4	11	10.1	Kmeans	/4	12	9.7
	/8	11	8.8		/8	12	10.9
	/16	10	8.9		/16	13	11.0
	/31	11	8.0		-	-	-
Sortedlist	/4	3	3.0	Vacation	/4	17	14.0
	/8	3	3.0		/8	20	16.0
	/16	3	3.0		/16	21	16.4
	/31	3	3.0		-	-	-

レッドが，再び，既に無効化された当該ライン上のデータへアクセスする場合には，L2 キャッシュからの無駄なデータ転送が必要となる．したがって，R/W テーブルによる管理対象とされたラインへのメモリアクセスの成功が，L1 キャッシュミスの増加の原因であると考えられる．

STAMP

Vacation では，FALSE STALL-NONXACT，および FALSE STALL-XACT を解消したことで実行サイクル数の削減を達成した．結果として Sortedlist 同様、各キャッシュ分割数の場合で false stall によるサイクル数がほぼ同量削減されていることが分かる．R/W テーブル上で管理される変数の配置されるアドレスを詳細に調査したところ，Vacation ではスレッドローカルに定義された異なる変数が，同一キャッシュライン上のある程度離れた位置に配置されており，これが FALSE STALL-NONXACT を発生させていることが分かった．

このようなデータ構造はある程度の大きさを有しているため，キャッシュラインサイズの 1/2 程度の大きさのブロックを最小単位とするだけで，誤った競合検出を抑制するのに十分であったと考えられる．

一方で，Kmeans では FALSE STALL-XACT の削減も認められたが，そもそも総実行サイクル数に占める割合が小さいため，削減されたことによる影響は少なかった．

6.4 ハードウェアコスト

提案手法を実現するため，R/W テーブルには，競合が発生したキャッシュラインの数だけのエントリが必要となる．そこで提案モデル (T_{16}) で実行した場合の各プログラムにおいて，R/W テーブル ひとつ当たりに使用されたエントリ数を調査した．この結果を**表 4** に示す．これより，最大で 21 エントリあれば，全てのプログラムにおいて R/W テーブルが溢れることなく false stall を防止することができることが分かる．ここで，R/W テーブルのひとつのエントリ当たりに必要となる Tag は，アドレス幅 $W = 64$ bit，キャッシュラインサイズ $L = 64$Bytes である場合，$W - \log_2 L = 64 - \log_2 64 = 58$ bit であり，また R，W^{my} および W^{other} はそれぞれ 16 bit である．したがって，1 つの R/W テーブルは幅 106 bit 深さ 21 行の RAM で構成でき，32 スレッドを実行可能な 32 コア構成のプロセッサの場合では R/W テーブルサイズの総和は約 8.9KBytes と少量である．

表 5 (T_{16}) における R/W テーブルエントリの総参照回数

GEMS		STAMP	
Prioque	4542.6	Kmeans	187.8
Sortedlist	97.4	Vacation	769.1

6.5 R/W テーブルエントリの参照コスト

5.3 節で述べた R/W テーブルエントリを参照する際には，レイテンシが発生する．そこで，そのオーバヘッドが性能にどの程度影響するかについて考察する．

まず，R/W テーブルエントリを参照した総回数を C，R/W テーブルエントリを 1 回参照するためのレイテンシを T とすると，その参照コストは $C \times T$ として概算できる．ここで，提案モデル (T_{16}) において 16 スレッドで実行した場合の R/W テーブルエントリの総参照回数を**表 5** に示す．また，6.4 節で述べた通り，R/W テーブルは約 8.9KBytes の RAM で構成できる．そこで，R/W テーブルエントリを L1 キャッシュと同レイテンシで参照できると仮定すると，本稿で用いたシミュレーション環境では $T = 3$cycels とおくことができる．これらより，総参照回数の最も多い Prioque についてそのコストを求めると，$4542.6 \times 3 =$ 約 1.3 万 cycles となる．一方, Prioque/16thrs の総実行サイクル数は約 600 万 cycle であるため，このオーバヘッドが総実行サイクル数に占める割合は約 0.002%と，僅かなものであることが分かった．よって今後は，R/W テーブルの利用効率を改善することで，本手法によるさらなる速度向上を見込めると考える．

7. おわりに

本稿では，既存の LogTM においてキャッシュライン単位で行っていた競合の検出操作を，R/W テーブルというハードウェアを保持することで細粒度化する手法を提案した．これにより，同一キャッシュライン上の異なる変数に対して，複数のスレッドがアクセスすることで発生する，本来競合ではないアクセスが競合として検出されてしまうことを防止した．

GEMS 付属の microbench 及び STAMP ベンチマークを用いてシミュレーションにより評価した結果，前述した競合の誤検出が防止できることを確認した．また，既存の LogTM に比べて，最大 89.4% ，16 スレッドにおいて平均 26.3%の実行サイクル数の削減が確認できた．

今後の課題としては，R/W テーブルの利用効率向上が挙げられる．R/W テーブルには過去に一度でも競合が発生したキャッシュラインに対応したエントリが登録されるが，以降一度もそのラインで競合が発生しない場合，エントリが R/W テーブルに残り続ける．このような今後参照されることのない無駄なエントリが増大することで，R/W テーブルがオーバーフローする可能性があるが，今回提案した手法では，トランザクションをアボートさせることで無駄なエントリを含む，全てのエントリを削除していた．そこで，そのような無駄なエントリの存在を判別し，定期的に R/W テーブルから削除する機構を追加することで，

R/W テーブルの利用効率を向上させることができると考えられる．

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] Shavit, N. and Touitou, D.: Software Transactional Memory, pp. 204–213 (1995).
[3] 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).
[4] Torrellas, J., Lam, M. S. and Hennessy, J. L.: False sharing and spatial locality in multiprocessor caches, *IEEE Transactions on Computers*, Vol. 43, pp. 651–663 (1994).
[5] Harris, T. L., Fraser, K. and Pratt, I. A.: A Practical Multi-word Compare-and-Swap Operation, *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pp. 265–279 (2002).
[6] Tabba, F., Hay, A. W. and Goodman, J. R.: Transactional conflict decoupling and value prediction, *Proceedings of the international conference on Supercomputing*, ICS '11, ACM, pp. 33–42 (2011).
[7] 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).
[8] 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).
[9] 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).
[10] 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).