Vol.2022-ARC-248 No.45
Vol.2022-SLDM-198 No.45
Vol.2022-EMB-59 No.45
2022/3/11

多様なトランザクショナルメモリ実装のスケーラブルなプログラムを用いた定量的比較

二本松 秀樹[1]　山本 和諒[1]　浅井 優太[1]　山下 淳[1]　塩谷 亮太[2]　五島 正裕[3]　津邑 公暁[1]

概要：ロックを補完・代替する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）が期待されている．今日までに多様な TM 実装が提案されているが，TM の活用は未だ一般に普及していない．その要因に，TM の性能を引き出すための知見が十分蓄積されていないことや，TM の性能が不十分であることが挙げられる．その点を解決するような研究として，仲池らは，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）の定量的比較を通じて，次世代 HTM 機構のあるべき姿について議論している．しかし，TM のソフトウェア実装であるソフトウェアトランザクショナルメモリ（Software Transactional Memory: STM）が評価対象に含まれていない上，さまざまな問題点が指摘されているベンチマークを使用した評価に基づいている点で，包括的な TM システムの再検討には十分ではない．
そこで本論文では，STM と HTM の両方を評価対象とし，さらに，性能がスケールするよう TM に適した形に改良を施したベンチマークを併用し，定量的比較を行う．TM の実装およびプログラムの差異が性能に与える影響を調査・解析し，TM の性能を引き出すプログラミングや TM システムが備えるべき機能・特徴を議論する．

1. はじめに

ロックを補完・代替する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）が提案されている．現在までに，HTM と STM のどちらもさまざまな実装が提案されており，商用プロセッサに HTM が搭載されている段階にきているが，TM の活用は未だ進んでいない．その理由として，TM の性能を引き出すための知見が十分蓄積されていないことや，TM の性能不足などが考えられる．そこで本研究では，HTM と STM の両方を対象とした包括的な TM 実装の比較および TM に適した形に改良を施したスケーラブルなプログラムを併用した TM 実装の定量的評価を行い，プログラマが TM を用いた並列プログラムをどのように記述すれば良いか，性能と生産性を両立するために TM 側がどのようなサポートをしていく必要があるかについて検討する．

以下，2 章では先行研究について取り上げ，3 章では本研究の対象である TM の概要，4 章では評価に使用する 2 つのベンチマークについて説明する．そして，5 章は HTM を使用するために必要なプログラム改修について説明し，6 章では評価を元に TM を解析する．最後に，7 章で TM を用いたプログラミングに必要な要件を検討し，8 章で結論を述べる．

2. 研究背景

1 章で述べたように，これまで数多くの TM 実装が提案されてきているが，それら STM と HTM を統一的に調査し，TM に対して評価を与える研究は十分に行われていない．Diegues ら [1] は複数の TM 実装の消費電力とスケーラビリティについて調査し比較を行っている．また，仲池ら [2] は，Intel 社のプロセッサに搭載されている HTM および IBM 社のプロセッサに搭載されている 3 つの HTM の，計 4 つの HTM の定量的比較を通じて．次世代の HTM 実装について議論している．

しかし，前者は Intel 社のプロセッサにおける評価にとどまっており，後者は STM との比較ができていない．さらに，HTM は 1 物理コアに対し 1 スレッドで最大性能を発揮するが，両研究で使用している Intel 社のプロセッサの物理コア数は 4 つであるため，性能面を平等に比較できるスレッド数は 4 スレッドに制限され，スケーラビリティについての調査も十分でない．また，評価に使用している

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 東京大学
The University of Tokyo
[3] 国立情報学研究所
National Institute of Informatics

Vol.2022-ARC-248 No.45
Vol.2022-SLDM-198 No.45
Vol.2022-EMB-59 No.45
2022/3/11

STAMP ベンチマーク [3] は，TM に適した形でプログラムが記述されていないという指摘もある [4].

そこで本研究では，TM の性能を引き出すように改良を施したベンチマークプログラムを STAMP と併せて使用し，HTM と STM の包括的比較を行う．TM 実装およびプログラム記述の差異と性能との関係を調査・解析することで，TM の性能を引き出すプログラミング，および TM システムが備えるべき機能や特徴について議論する．

3. Transactional Memory

3.1 Transactional Memory の概要

ロックに代わる並行性制御機構として，TM が提案されている．TM はデータベースの更新・検索操作に用いられる概念であるトランザクション処理をメモリアクセスに応用した機構である．TM を使用する場合，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含む一連の処理をトランザクション（以後，Tx と省略する場合がある）として定義し，これらを投機的に並列実行することで，ロックを使用する場合よりも高い並列度が得られる．また，プログラマは，トランザクションを定義する際に，トランザクションの開始と終了を記述するだけでよく，デッドロックなどを意識する必要がない．したがって，TM は粗粒度ロックと同等以上の高いプログラマビリティを持ちつつ，細粒度ロックと同等以上の高い性能を発揮するポテンシャルを有しており，生産性と性能を両立しうる機構として期待されている．

TM は，専用ハードウェアを用いて実装するか否かで 2 種類に大別される．1 つは専用ハードウェアを用いて実装された HTM であり，もう 1 つはソフトウェアのみを用いて実装された**ソフトウェアトランザクショナルメモリ（Software Transactional Memory: STM）**[5] である．HTM は，専用ハードウェアによって，トランザクションを並列実行するために必要な操作のオーバヘッドを抑制できる．一方，専用ハードウェアが実装されていない環境ではプログラムを実行できないため，プログラムの移植性が低い．また，専用ハードウェアに起因する制約により，トランザクションを実行できない場合がある．STM は，ソフトウェアのみで実装された TM であり，トランザクションを並列実行するために必要な操作のオーバヘッドが大きくなる．一方，専用ハードウェアを必要としないため，現在広く普及している汎用プロセッサ上で動作可能であり，高い可用性を持つ．

3.2 競合検出

TM では，投機的にトランザクションを実行するために，共有変数へのアクセスを監視する．そして，トランザクションを実行している複数のスレッドが同一共有変数へアクセスを試みた場合，**競合（Conflict）**を検出する．

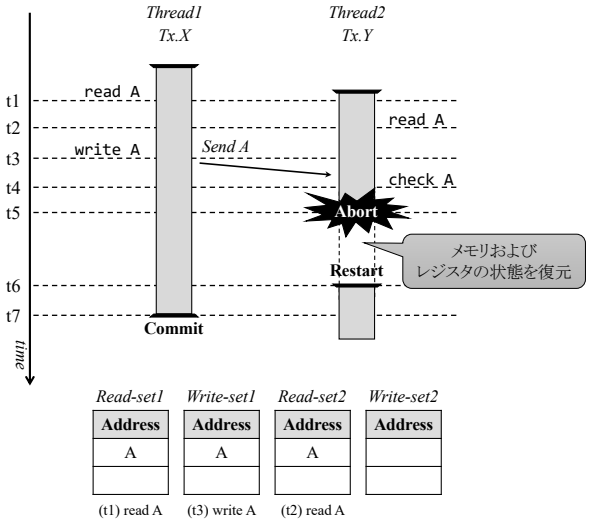


図 1 競合検出の動作

ここで，トランザクションの並列実行および競合検出の動作を**図 1** に示す．図におけるバーはそれぞれトランザクション処理を示しており，下向きに時間軸をとっている．図中の A，B は共有変数のアドレスを表し，`read A`，`read B` はそれぞれアドレス A およびアドレス B に対するロード命令を，`write A`，`write B` はそれぞれアドレス A およびアドレス B に対するストア命令を，当該時刻において発行したことを表している．また，図の下部に示す Read-set および Write-set は変数のアドレスを記憶するために各スレッドに関連付けられている専用領域を表す．

まず，*Thread*1 が `read A` を実行したとする（t1）．このとき，*Thread*1 はアドレス A を *Read-set*1 に記憶する（t1）．その後，*Thread*2 は `read A` を実行し，アドレス A を *Read-set*2 に記憶する（t2）．次に，*Thread*1 が `write A` を試みたとする（t3）．このとき，*Thread*1 は書き込み対象のアドレスであるアドレス A を *Write-set*1 に記憶し（t3），*Thread*2 に対してアドレス A を送信する．*Thread*2 は，受信したアドレスと自身の専用領域に記憶されたアドレスとを比較する（t4）．この例では，*Thread*2 がトランザクション内で `read` したアドレス A に対する `write` であるため，このアクセスを競合として検出する．アクセス競合を検出した場合，一貫性を保証するため，いずれかのトランザクションが実行を中止する．これを**アボート（Abort）**という．どちらのトランザクションがアボートするかは実装により異なるが，ここでは *Thread*2 の *Tx.Y* がアボートし，*Thread*1 の *Tx.X* が処理を続行したとする（t5）．トランザクションをアボートしたスレッドは，トランザクションの開始時点におけるメモリおよびレジスタの状態を復元する．この処理を**ロールバック（Rollback）**という．ロールバックを終えると，トランザクションの開始時点から処理を再実行する．この例では，*Thread*2 が *Tx.Y* 開始時点の状態を復元し，*Tx.Y* を再実行する（t6）．一方，処理を続

行する *Thread*1 は，*Tx.X* の終了までに他の競合が検出されず，トランザクションの処理を最後まで完了した場合，トランザクション内で行ったデータの更新を確定できる（t7）．これを**コミット（Commit）**という．

また，競合検出は，アクセス競合が発生しているか否かを検査するタイミングによって，以下の 2 つに大別される．

Eager Conflict Detection（EagerCD）：トランザクション内でメモリアクセスが発生する度に，そのアクセスにより競合が発生するか否かを検査する．

Lazy Conflict Detection（LazyCD）：トランザクションのコミットを試みる時点で，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るか否かを検査する．

EagerCD では，競合が発生した際に即座にそれを検出できるのに対し，LazyCD ではコミット時まで競合を検出することができない．そのため，EagerCD ではアボート時にその時点までのトランザクション実行が無駄になるだけであるが，LazyCD ではトランザクション全体の実行が無駄となってしまう．しかし，LazyCD ではメモリアクセスの度に競合を検出する必要がないため，競合が発生しない場合には EagerCD よりオーバヘッドが小さく済む．

3.3 HTM における競合以外のアボート理由

前節では，競合によりトランザクションがアボートすることについて述べたが，HTM においてトランザクションがアボートする原因は，競合だけではない．競合以外の原因でアボートする場合は大きく分けて 3 つある．

1 つ目は，トランザクション内でアクセスするアドレス数が，専用ハードウェアで記憶可能なアドレス数の上限を超える場合である．競合検出のためのアクセス対象アドレスの記憶は，専用領域を用いて行うが，この領域の容量は当然ながら有限である．この上限を超える数のアクセスが発生する場合，一貫性を保証できなくなるため，トランザクションをアボートする．以降，本論文では，このようなアボートを**キャパシティアボート**と呼ぶ．

2 つ目は，トランザクションを実行している途中で割込みが発生する場合である．この場合，Intel 社のプロセッサでは，一貫性を保証できなくなるため，トランザクションをアボートする．一方，IBM 社の POWER プロセッサでは，suspend および resume と呼ばれる機能が実装されており，割込みが発生しても，トランザクションをアボートしないことが可能になっている．

3 つ目は，トランザクションとして定義した区間内に TM でサポートされていない処理が含まれる場合である．例えば，入出力処理やキャッシュラインをフラッシュする処理が挙げられる．トランザクション内で入出力処理が発生した場合，その後の処理が原因でトランザクションをアボートしても，入出力した結果を破棄し，ロールバックすることができない．したがって，TM では printf 関数のような入出力を行う関数がトランザクション内に含まれる場合，トランザクションをアボートする．

また，トランザクション内でキャッシュラインをフラッシュする命令が使用されると，当該ラインで記憶していたアドレスに対する競合を検出する前に，トランザクション内で行った更新をメモリに反映してしまうため，トランザクションをアボートする．

これら 3 つのケースでトランザクションをアボートした場合，ロールバック後にトランザクションを再実行しても，再び同じ原因でアボートするため，処理を進行できなくなる．そのため，TM を使用してトランザクションを実行することが不可能となり，別の方法を用いてトランザクションとして定義した区間の処理を進行する必要がある．このように，別の方法を用いた実行に切り替えることを**フォールバック（Fallback）**という．フォールバック先として，一般にロックによる排他実行が用いられる．この場合，トランザクションとして定義した区間を排他実行するため，フォールバックする際に他のトランザクションを全てアボートする必要がある．したがって，フォールバックもトランザクションのアボートを引き起こす原因となる．

4. 評価に使用するベンチマーク

4.1 STAMP と Stampede

3 章で示したような動作を実現するために，今日までにさまざまな TM 実装が提案されており，それらの TM を評価するベンチマークとして STAMP ベンチマーク [3] が一般的に使用されている．しかし，その STAMP のプログラムで採用されているデータ構造とプログラミングモデルには問題があるという主張があり [4]，STAMP を改良したベンチマークとして Stampede が提案されている．

4.2 2 つの改良原則

Stampede は，以下に示す 2 つの原則に則って STAMP を改良している．

Disjoint Accesses – アクセスの分離

TM は，異なるスレッドが同一のアドレスにアクセスすると競合が発生したとみなし，トランザクションをアボートさせる．従って，スレッドがアクセスするデータを可能な限り分割し，不必要なアボートを抑制する必要がある．例えば，STAMP で使用されているデータ構造の赤黒木を考える．赤黒木に含まれるあるノードに対してアクセスを行う際，赤黒木の仕組み上，ルートノードから目的のノードまで順に辿る．その間に経由するノードに対して別のスレッドがアクセスを試みると，赤黒木の仕組み上，経由したノードへのアクセスが重複する．各スレッドが最終的にアクセス対象とするノードが異なっているにもかかわらず，競合

Vol.2022-ARC-248 No.45
Vol.2022-SLDM-198 No.45
Vol.2022-EMB-59 No.45
2022/3/11

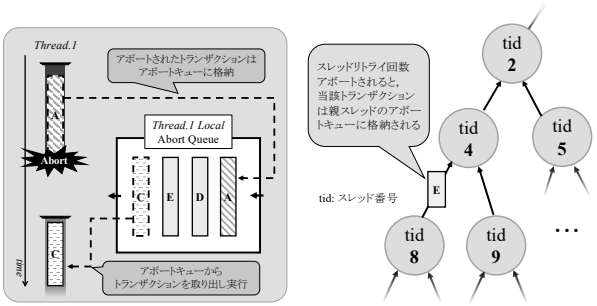
が発生し，どちらか一方のスレッドはトランザクションをアボートさせる必要がある．このように，使用するデータ構造によっては，不必要な競合が発生する．たとえば赤黒木の代わりにハッシュ表を用いることにより，処理対象データが異なっている場合に，アクセス自体も分離される．

さらに，全スレッドで 1 つのハッシュ表を共有するのではなく，各スレッドにローカルなハッシュ表を用意し処理終了時にそれらをリダクションしたり，動的なメモリ確保は OS による調停がボトルネックになるため，スレッドごとに別の領域からメモリを確保するメモリアロケータに変更したりして，他のスレッドおよび OS とのやりとりを抑制するように STAMP に修正を加えている．

Virtualized Transactions – トランザクションの仮想化

一般的に，トランザクションがアボートされると，同一のトランザクションを再実行する処理が行われる．このような実行は，スレッドとトランザクションが強く結びついた状態とみなすことができる．これに対し Stampede では，スレッドからトランザクションを切り離すことができるよう STAMP に対して修正を加えて，トランザクションがアボートされると代わりに別のトランザクションを実行したり，他のスレッドにトランザクションの実行を委譲することが可能となっている．本論文では，以後，スレッドから切り離して実行可能なトランザクションを，**仮想化された**トランザクションと呼称する．仮想化されたトランザクションを実行中に，何らかの理由でアボートさせられた場合，図 2 の (a) に示すように，スレッドごとに設置されているアボートキューに対して，アボートさせられたトランザクションを格納する．その後，キューから別のトランザクションを取り出して実行する．

これに加え Stampede は，トランザクションを徐々に直列化する **Serialization Tree** という機能を提供している．図 2 の (b) に示すように，それぞれのスレッドには 0 から始まる連番（以下，tid とする）が割り振られ，各スレッドが 2 分木のように関連づけられている．あるトランザクションが，予め設定された閾値（以後，スレッドリトライ回数と呼称する）を超えた回数アボートさせられると，当該トランザクションを実行しているスレッドが持つ tid 番号の半分に当たる tid 番号を持つスレッド（以後，親スレッドと呼称する）のアボートキューに当該トランザクションを格納する．アボート回数の多いトランザクションほど親スレッドに引き渡されていき，高頻度でアボートするトランザクションが，同一のアボートキューの中に存在する可能性が高まっていく．アボートキューの中身は一つづつ取り出されて実行されるため，競合が発生する可能性の高いトランザクションほど同時刻に処理されづらくなる．



(a) 仮想化されたトランザクションの動作　(b) Serialization Tree スレッド同士の関係

図 2　Virtualized Transactions の概要

この改良原則によって，トランザクション同士の競合が連続する場合に対処しており，2 つ以上のスレッドが連続して互いにトランザクションをアボートさせあい処理の進行が妨げられる**ライブロック（Live Lock）**を回避できる可能性が高まるほか，トランザクションのスケジューリングを工夫することによって競合発生率の更なる低減の可能性を残している．

4.3　Stampede の特筆すべき特性

Stampede[4] を提案している Nguyen らは，従来 TM が担ってきた役割の一部をアプリケーション側に委譲することで，TM の実装を軽量化でき，かつ，性能を向上できると主張している．また，STAMP からのソースコードの改変は 12%ほどで，これは無理のない改変であると言及している．ところが実際は，4.2 節や本節で示すような Stampede に施された改良は，C++で記述された並列プログラミング向けライブラリ Galois を用いて行われており，総コード量は STAMP と比較して大幅に増加している．

4.3.1　EagerCD を前提とした STM 向けアプリケーションコード

STM を用いて Stampede を実行する場合，その STM はアクセス時に競合検出を行う EagerCD でなければならない．コミット前まで競合検出を遅延させる LazyCD な STM を用いることができない理由は，保護対象となる変数の *Read-set* と *Write-set* を作成せず，ロールバックしない実行形態をとるためである．図 3 に，Stampede で定義されているトランザクションで行う処理が，STM および HTM を用いた場合にどう異なるかについて示す．Stampede を STM を用いて動作させる際，トランザクション開始時にトランザクション内でアクセス予定の変数へのアクセスが完全に排他的になるよう，ロック獲得フェーズで必要なロックを獲得したのち，処理範囲の実行を開始する．獲得する予定のロックのうち一つでも他のスレッドによって獲得されていることを STM の機構で検出すると，実行中のトラン

Vol.2022-ARC-248 No.45
Vol.2022-SLDM-198 No.45
Vol.2022-EMB-59 No.45
2022/3/11

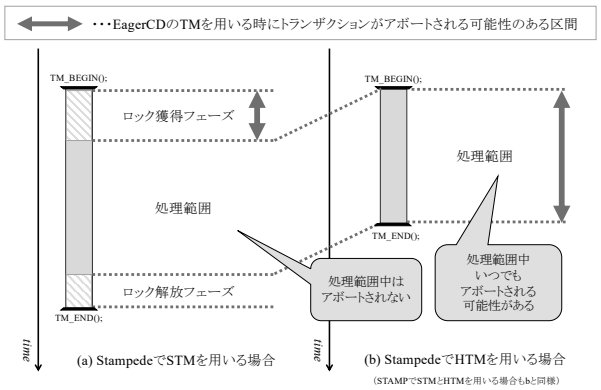


図 3 Stampede でトランザクションとして定義した区間で行う処理の違い

ザクションをアボートさせる．ロールバックはその時点で獲得済のロックに対してのみ行われ，その後，Virtualized Transactions の原則に則り，別のトランザクションの実行を試みる．このようにして，処理範囲で行われる変数へのアクセスを完全に排他的に行うことを保証し，代わりに，STM の *Read-set* と *Write-set* による変数の保護を省略している．LazyCD な STM を利用する場合，コミット前まで競合検出を遅延することになるが，ロールバックを試みても，保護対象である変数の *Write-set* を作成していないため適切にロールバックできず，データの一貫性を保証できない．

以上のように示した実行形態の実現は，プログラマが適切な粒度でロックを定義することを前提としている．なお，図 3 の (b) で示すように，Stampede を HTM を用いて実行する際には本ロックを使用することはなく，全ての競合検出とロールバックは HTM に委託されている．これによって，HTM を用いる場合には，処理範囲のどのタイミングでもトランザクションがアボートされる可能性がある．

なお，本実行形態は TM を用いない細粒度ロックによる実行とは異なっている．TM を用いない細粒度ロックによる実行では，ロックの獲得に失敗するとロックの解放まで待機する必要がある一方で，本実行形態は，使用予定の変数のロックの獲得ができない場合，別のトランザクションの実行に成功することができれば高並列な実行が可能である．

4.3.2 アプリケーション側での補助的 *Write-set*

トランザクションの開始時にはアクセス対象と判明せず，トランザクション実行中にアクセス対象と判明するような変数が出現する場合に備え，一部プログラム（bayes, yada）では，STM の *Write-set* とは別に，アボート時のロールバック用ログをアプリケーション側で用意している．トランザクションがアボートされると，そのロールバック用ログを使用して，トランザクション実行前の値にロールバックする．4.3.1 項と同様 Stampede を HTM を用いて実行する際には，本項で示した補助的 *Write-set* を使用することはなく，*Write-set* の作成は HTM に委託されている．

5. Best-effort 型 HTM に向けた Stampede プログラムの改変

5.1 Best-effort 型 HTM とフォールバック

3.3 節で述べたように，HTM を用いる場合，ハードウェアの資源制約によってコミットできないトランザクションがある．よって，そのようなトランザクションの実行を適切に完了させ，プログラムの進行を保証する必要がある．IBM 社の BlueGene/Q[6] に搭載されている HTM は，ハードウェアの資源制約によってコミットできないトランザクションであっても．その実行を完了させることができる仕組みを備えている．一方で，今回の評価に使用するプロセッサの HTM 機構はいずれも，そのようなトランザクションの実行を完了する仕組みを備えておらず，プログラマが，HTM で完了できない場合に対処しておく必要がある．このような HTM を **Best-Effort 型 HTM** と呼ぶ．一般には，アボートの発生およびその原因が HTM から提示されるため，ロックなどを用いた実行をフォールバックパスとして定義しておき，アボート原因に基づいて適宜そのフォールバックパスに切り替えるなどの処理をプログラマが記述しておくことで対処する．

Stampede は IBM 社の BlueGene/Q に搭載されている HTM を使用した評価 [4] がなされている．しかし，Best-effort 型 HTM を使用した実行の評価は行われていないため，Stampede のコードでは，HTM でコミットできないトランザクションを完了させるための対処がなされていない．そこで，Stampede のプログラムを Best-effort 型 HTM を用いて動作させることができるよう，ロックを用いたフォールバックパスを実装した．

5.2 Virtualized Transactions に配慮したフォールバックパス実装

HTM を使用してプログラムを動作させる際，フォールバックパスに移行すべき場面は次の 2 つに大別され，それぞれの場合で再実行方針が異なる．

Case1）Tx を再実行しても再びアボートすることが確実視される場合

キャパシティアボートが発生した場合および非サポート命令を使用した場合が該当する．このような場合には，再実行するよりも，フォールバックパスに即座に移行するのが望ましい．

Case2）トランザクション同士が過度に競合したと判定される場合

競合によるアボートの場合は，再実行によってトランザクションをコミットできる可能性がある．Case1 と同じようにフォールバックパスへ即座に移行すると HTM を用いた並列実行でコミットできたはずの機会

を逃してしまうため，トランザクションを何度か再実行させるのが望ましい．しかし，再実行回数に制限を設けず無限に再実行を繰り返すと，トランザクション同士の競合を解決できないライブロック（Live Lock）などが発生しプログラムの正常な進行を妨げる．従って，競合によってアボートした回数を記録し，予め定めた閾値を超えた場合に，ロックを用いたフォールバックに移行するのが望ましい．

以上のように，アボートした理由とその回数に応じて，ロックを用いたフォールバックパスに移行するタイミングを制御する必要がある．

今回 Stampede に含まれるプログラムに対して施したフォールバックパスは仲池らの実装をベースにしている．しかし，4.2 節で示したように，Stampede ではトランザクション同士の競合を Virtualized Transactions の原則を用いて解決している．そこで Virtualized Transactions の原則を可能な限り用いるよう配慮したロックフォールバックパスを実装した．

まず，アボート回数を記録するカウンタをトランザクションごとに設けた．設置したカウンタは次の 3 つである．

persistent-retry counter ：ハードウェア資源制約によるアボート回数を記録するカウンタ

transient-retry counter ：トランザクションの競合によるアボート回数を記録するカウンタ

lock-retry coutner ：ロックによるアボート回数を記録するカウンタ

アボートキューに当該トランザクションを格納する際に，アボート理由に対応するカウンタをインクリメントする．そして，アボートキューから取り出したトランザクションのカウンタが閾値を超えていた場合，ロックを用いたフォールバックパスに移行してトランザクションを実行するよう Stampede に含まれるプログラムを修正した．

なお，これらのアボート理由の判別には，HTM が提供する機能を利用している．トランザクションをアボートした際に，POWER プロセッサではトランザクションの状態を示すレジスタ（Transaction EXception And Status Register: TEXASR），Intel 社のプロセッサでは EAX レジスタ，の特定のビットを利用し，再実行によってトランザクションを実行完了可能かどうか，を提示する．これを利用し，再実行によってトランザクションの実行を完了できる場合 *transient-retry counter* を，実行を完了できないと提示してきた場合，*persistent-retry counter* をインクリメントする．

6. 評価

6.1 評価環境

今回，5 種類の STM と 2 種類の HTM を使用して評価する．STM は TL2[7], NOrec[8], TinySTM[9], SwissTM[10] および XTM[4]，HTM は Intel 社のプロセッサおよび IBM 社の POWER プロセッサに搭載されている HTM を使用した．評価に使用したプロセッサとコンパイラを表 1 に示す．また，これらの Best-effort 型 HTM を用いる際のフォールバックパスに移行する閾値を表 2 に示す．

表 1 評価環境

CPU	Xeon Gold 6152	POWER9
物理/論理 コア数	22/44	16/64
クロック	2.10GHz	2.20GHz
L1 キャッシュ	32KB	32KB
L2 キャッシュ	1MB	512KB
L3 キャッシュ	30.25MB	10MB
メモリ	16GB	16GB
OS	ubuntu 16.04.5 LTS	ubuntu 18.04.2 LTS
コンパイラ	gcc version 7.5.0	gcc version 7.4.0
オプション	-O2	-O2

表 2 ロックを用いたフォールバックパスに移行する閾値

	STAMP	Stampede
persistent-retry counter	1 回	2 回
transient-retry counter	16 回	$\log_2(\#\text{threads}) \times 7$
lock-retry coutner	16 回	16 回

STAMP における閾値は，仲池ら [2] の設定に準じた回数になっている．一方，Stampede の *transient-retry counter* に関する閾値について，仲池らが設定した数値では，Serialization Tree によるトランザクションの漸進的な直列化を待たずにロックを用いたフォールバックパスに切り替わってしまい，Serialization Tree の影響を正確に評価できない．そこで，*transient-retry counter* の閾値を Serialization Tree の高さ $\log_2(\#\text{threads})$ にスレッドリトライ回数 7 を乗じた値に設定した．この閾値によって，Serialization Tree の末端で試行を開始したトランザクションが Serialization Tree を移動し，0 番目の tid を持つスレッドに到達できるようになる．なお，スレッドリトライ回数 7 は Stampede のデフォルト値から変更していない．

6.2 速度向上率とフォールバック実行割合

まず，STAMP を 16 スレッドで実行した際の速度向上率を図 4 に，Stampede を 16 スレッドで実行した際の速度向上率を図 5 に示す．また，Intel 社のプロセッサおよび IBM 社のプロセッサにおいて，スレッド数を増減させた際の速度向上率の変化を図 6 および図 7 に示している．なお，本論文において速度向上率とは，TM を使用せず STAMP を 1 スレッドで逐次実行した場合の実行時間に対する，TM を使用して複数スレッドで実行した際の実行時間の比を表し，TM を使用せず STAMP を 1 スレッドで実行した場合の実行時間は表 6.2 に示す数値を使用している．Stampede の速度向上率も STAMP の逐次実行の実行時間をベースに

表 3 速度向上率のベースとなる各プログラムの 1 スレッドの実行時間

	STAMP Intel	STAMP Power
bayes	10.682	8.447
genome	6.587	4.743
intruder	16.874	15.621
kmeans-low	13.030	16.013
kmeans-high	2.084	2.436
labyrinth	76.851	36.181
ssca2	14.786	11.396
vacation-low	13.521	16.397
vacation-high	18.111	22.400
yada	7.755	7.553

しているのは，STAMP と Stampede の間にコードの差異はあるものの Stampede は STAMP で行っている処理内容を踏襲しているためである．なお，Stampede の kmeans は本評価環境において，特異に実行時間が遅い問題が発生しているため，評価の対象にしていない．

まず，labyrinth および yada のプログラムにおいて，STAMP と Stampede のいずれも，HTM を使用した場合の速度向上率が 1 を大きく下回っている．これは，両プログラムはトランザクション中にアクセスする変数の数が多く，キャパシティアボートが多発することで，ロックを用いたフォールバックパスに移行しているからである．ここで，HTM を用いて各プログラムを実行した際に，トランザクションとして定義した区間の実行をロックを用いたフォールバックにより完了した割合を図 8 に示す．図中の凡例が示す項目は，フォールバックパスへ移行する要因となったカウンタ名を表している．例えば，*persistent-retry counter* の閾値超過によってフォールバックパスに移行した場合は，*persistent-retry counter* の割合に含まれる．

グラフの labyrinth および yada の結果より，トランザクションのほとんどをフォールバックパスによって完了していることが確認できる．この二つのプログラムに含まれるトランザクションでは，多量のデータにアクセスしており，そのようなトランザクションではハードウェアの容量を超えて *Read-set* や *Write-set* を作成することはできないため，必要があればフォールバックパスへの移行を即座に行う必要がある．一方，ロックを用いるフォールバックパスへの過度な移行はトランザクションの並列度を低下させる要因となるため，グローバルロックを用いる実行は可能な限り回避されるべきである．

6.3 HTM よりも高速に動作する STM の実行の分析

Nguyen ら [4] は，BlueGene/Q に搭載されている HTM のみを用いて Stampede を評価している．Virtualized Transactions の原則はアボートされた後別のトランザクションに切り替える処理を行うことで実現可能であるが，BlueGene/Q に搭載されている HTM は，トランザクションがアボートされたことをプログラムに対して提示せず，プログラマがアボートされた後の処理を記述できない仕様であるため，Virtualized Transactions による影響を評価できていない．一方，本研究で使用している Intel 社のプロセッサおよび IBM 社の POWER プロセッサに搭載されている HTM は，トランザクションがアボートされたことをプログラムに対して提示し，アボートされた後の処理をプログラマに委ねる．この仕組みを使用して，Virtualized Transactions を HTM を用いた場合にも実現することで，STM と HTM を同列に比較することを可能にした．

図 5 より，Stampede における STM と HTM の速度向上率を比較すると，HTM の速度向上率が STM と比較して劣っていることがわかる．一般に，先述した labyrinth および yada のように HTM を用いてトランザクションをコミットできない場合を除き，HTM の速度向上率が STM の速度向上率を上回ることが知られている．ハードウェアによるサポートを受ける HTM では，競合検出やロールバックに関わる負荷が小さいためである．実際，図 4 に示した STAMP の速度向上率では，intruder，kmeans, ssca2, vacation などで HTM の速度向上率が STM の速度向上率を上回っていることが確認できる．ところが図 5 からは，この傾向が必ずしも Stampede には当てはまらないことがわかる．この原因を調査したところ，STAMP ではほとんどフォールバックパスに移行しないにもかかわらず，Stampede ではフォールバックパスに移行した割合が高いプログラムが存在していることがわかった．図 8 より，STAMP と Stampede とを比較すると，複数のプログラムにおいて，フォールバックパスへの移行の原因のうち，*persistent-retry counter* の閾値超過による割合が大幅に変化している．特に，ssca2 が顕著であり，STAMP ではほとんどのトランザクションをフォールバックパスに移行せず完了できているのに対し，Stampede では Intel で 66%，POWER で 89%のトランザクションがフォールバックパスに移行している．この原因を調査したところ，2 つ原因が判明した．以下，判明した原因を順に述べる．

定義区間が変更されているトランザクション

Stampede では，STAMP と比較して大幅に定義区間が変更されているトランザクションが複数存在していることが判明した．STAMP ではトランザクションに含まれていない処理を，Stampede ではトランザクションに含むよう変更されている場合や，STAMP では複数に分けて定義されていたトランザクションが Stampede では一つのトランザクションに統合されている場合などが存在していた．

そのような変更が施されているプログラムのひとつに intruder がある．intruder はネットワーク上に流れるパケットを監視するシステムをエミュレートしたアプリケーションで，STAMP では以下のような順で処理が進行する:
(1) ネットワークストリームを模したキューの操作

Vol.2022-ARC-248 No.45
Vol.2022-SLDM-198 No.45
Vol.2022-EMB-59 No.45
2022/3/11

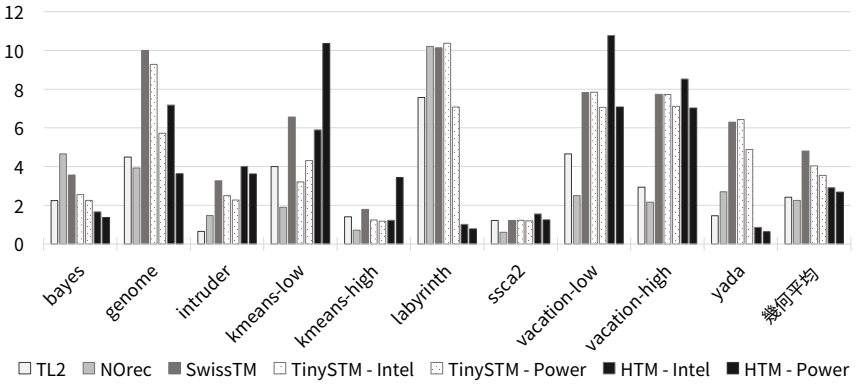


図 4　STAMP の速度向上率 (16 スレッド)

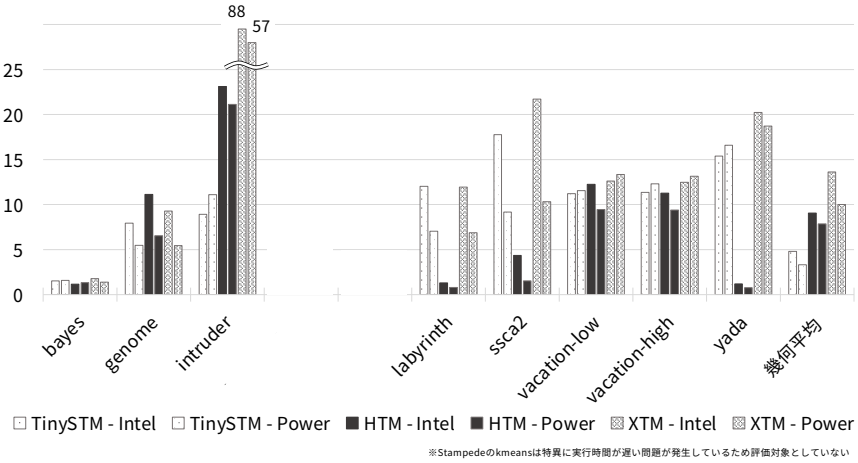


図 5　Stampede の速度向上率 (16 スレッド)

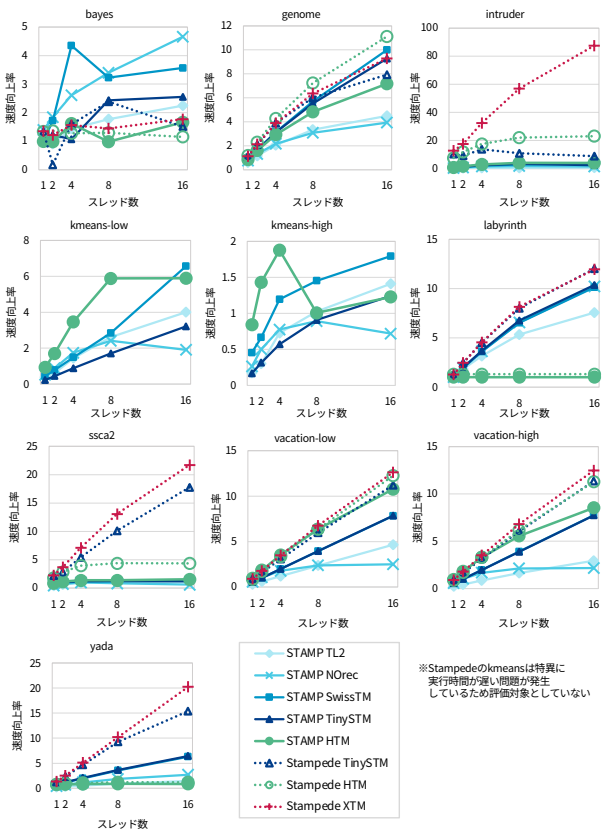


図 6　Intel Xeon プロセッサを用いた場合の速度向上率の変化

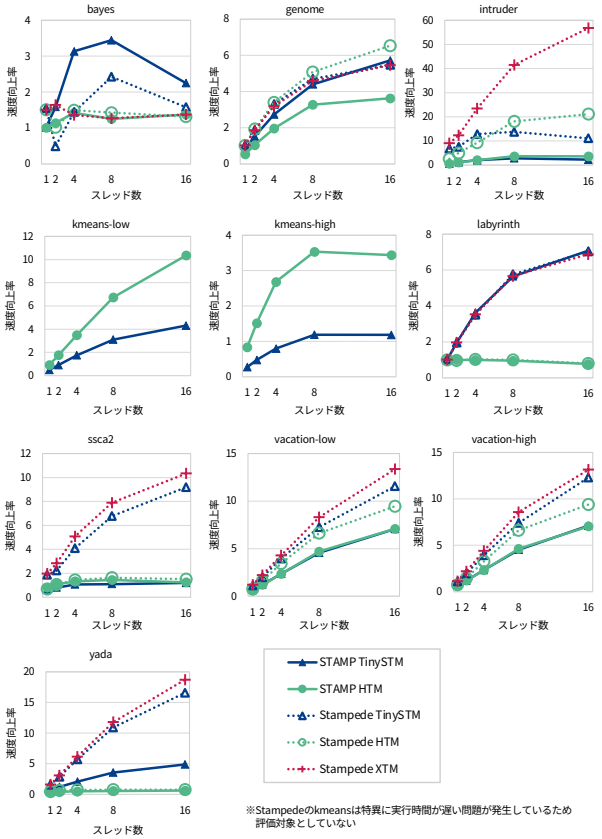


図 7　IBM POWER プロセッサを用いた場合の速度向上率の変化

Vol.2022-ARC-248 No.45
Vol.2022-SLDM-198 No.45
Vol.2022-EMB-59 No.45
2022/3/11

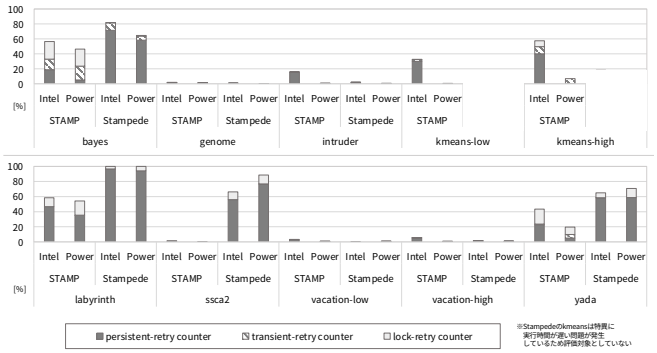


図 8 HTM を用いた場合のフォールバックパス実行割合 (16 スレッド)

(2) 複数のパケットをデコード（再構成）する処理
(3) 再構成が終了したデータとシグネチャとの比較

STAMP では，1 と 2 の処理がそれぞれトランザクションとして定義されており，3 の処理はトランザクションとして定義されずスレッドローカルな領域で処理される．一方，Stampede では，1 の処理はトランザクションとして定義されておらず，並列処理ライブラリである Galois を使う形に効率化されている．また，2 と 3 の処理がまとめて一つのトランザクションとして定義されている．3 の処理は，トランザクションとして定義される必要がない処理であるが，Stampede ではトランザクションの定義に含まれている．このような修正は，vacation を除いた他のプログラムでもみられる．定義範囲が拡大されたトランザクションを実行しようとすると，STAMP よりも多数の変数にアクセスするため，アクセスを記憶するために必要となる *Read-set* と *Write-set* の容量を圧迫する．そのため，今回使用した Intel や POWER の HTM を使用して Stampede を実行すると，STAMP ではキャパシティアボートに至らないようなプログラムであっても Stampede ではキャパシティアボートする場合があり，その結果として，*Read-set* や *Write-set* の容量に制約のない STM と比較して速度向上率が劣ったと考えられる．

ロック獲得フェーズの効果

Stampede では，ロック獲得フェーズが STM を用いる際の性能に大きく影響していると判明した．表 4 は，図 3 の (a) に示した形態のトランザクションを XTM で実行した結果と，(b) に示した形態のトランザクションを HTM で実行した結果を示している．これらの主な差はトランザクション開始時のロック獲得フェーズの有無であり，ロック獲得フェーズを用いる XTM の方が実行時間と総アボート回数の両方が優れた結果となっている．従って，ロック獲得フェーズによってなされる早期の競合検出は有効に働いているということがわかる．

表 4 Stampede の vacation のプログラムを 16 スレッドで実行した際の実行結果

	XTM Tx 仮想化およびロック獲得	Intel HTM Tx の仮想化
実行時間 [秒]	1.45	1.95
Tx 初回試行数	4,194,304	4,194,304
総アボート回数	1,318,455	3,207,922
キャパシティアボート回数	-	63,257
データ競合によるアボート回数	1,318,455	125,707

7. 性能と生産性を両立する TM プログラミングとそれをサポートする TM に求められる要件

TM では一般に，トランザクションがアボートされた場合は同一のトランザクションの再実行を試みるスケジューリングが採用されている。このようなスケジューリングでは，Stampede で行われているような，トランザクション開始時にロックを用いて競合を検出する方法を採用したとしても，再実行により再度競合が検出されるのみであるため，競合を回避するには，再実行までの待機時間調整など，スケジューリングの付加的な工夫が必要であった。これに対し Stampede では，競合検出時に異なるトランザクションの実行に切り替えることができる Virtualized Transactions が採用されており，これが，トランザクション開始時に競合を検出する方法と非常に親和性があり，高い性能が達成できた，

また，このトランザクション開始時に競合検出するためのロック獲得フェーズは，アプリケーション毎に個別に実装する必要がある．このアプリケーションレベルで実装されたロック変数は，STM を用いて競合検出を行うための要件でもあるため，アプリケーションプログラマに，的確な粒度で過不足なくロックを定義することが要求されることは，TM を用いた並列プログラミングが元来もっている高い生産性が大きく損なわれることを意味する．Nguyen らは，このような最適化を妥当であると主張しているが，高速且つ完全な TM の動作のための責任をアプリケーションプログラマに担わせることは現実的ではなく，競合検出の複雑さを TM システム側で吸収し，なおかつ，高速なトランザクション実行を実現できるのが望ましい．

生産性の面を考慮すると，4.3 節で示したようなトランザクションにおける変数の保護は，TM に担わせるのが望ましい．また，STM を用いて Stampede を動作させる際のロック獲得フェーズで行われるような早期の競合検出を，これまで研究されてきた競合予測技術に置換するのは有効であると考える．例えば，Diegues ら [11] によって考案さ

れた技術は，HTM を用いる際のトランザクション外での競合予測を可能にしている．ロック獲得フェーズを HTM のトランザクションでも導入すると，ロック変数が *Read-set* および *Write-set* の容量を圧迫し，意図せぬキャパシティアボート発生の要因となる．従って，容量制約のある TM で早期の競合検出を導入する場合，その競合検出はトランザクション外で行われるのが望ましく，Diegues らが提案しているような競合予測技術はそれを満たしている．

これら技術と，高速な競合検出およびロールバックの両方を備える TM システムを構築すれば，性能と生産性を両立した並列プログラミングの実現に近づくと考えられる．

8. 結論

本論文では，複数の TM 実装を用いて STAMP ベンチマークおよび Stampede ベンチマークを動作させ，TM を定量的に評価した．評価の結果，Stampede のプログラムと XTM の組み合わせが，性能面で最も優れていることを確認した．そこで，Virtualized Transactions の原則および XTM を詳しく分析したところ，トランザクションの開始時に行う早期の競合検出と Virtualized Transactions との組み合わせが有効に働いていると判明した．しかし，その Virtualized Transactions と早期の競合検出は，アプリケーションプログラマに負担を強いることにより実現されている．

そこで，プログラマの負担を減らしながら，Virtualized Transactions および XTM の組み合わせに比肩する性能を実現する方法を考える必要がある．本論文ではこの方法について，初期的な検討および考察を行なった．今後は更に考察を進め，TM システムと TM プログラミングの理想的な役割分担を検討しながら，TM システムが持つべき機能を追求し，その実装・評価を通じて，生産性と性能を両立する並列プログラミング環境の実現を目指す。

謝辞 本研究の一部は，JSPS 科研費 21H03408 の助成を受けたものである．

参考文献

[1] Diegues, N., Romano, P. and Rodrigues, L.: Virtues and limitations of commodity hardware transactional memory, *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 3–14 (2014).
[2] Nakaike, T., Odaira, R., Gaudet, M., Michael, M. M. and Tomari, H.: Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8, *ACM SIGARCH Computer Architecture News*, Vol. 43, No. 3S, pp. 144–157 (2015).
[3] Minh, C. C. et al.: STAMP: Stanford Transactional Applications for Multi-Processing, *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08)* (2008).
[4] Nguyen, D. and Pingali, K.: What Scalable Programs Need from Transactional Memory, *Proc. 22nd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 105–118 (2017).
[5] Shavit, N. and Touitou, D.: Software Transactional Memory, *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pp. 204–213 (1995).
[6] International Business Machines Corporation: *IBM System BlueGene Solution BlueGene/Q Application Development*, 1 edition (2012).
[7] Dice, D., Shalev, O. and Shavit, N.: Transactional Locking II, *Proc. 20th Int'l Conf. on Distributed Computing (DISC'06)*, pp. 194–208 (2006).
[8] Dalessandro, L., Spear, M. F. and Scott, M. L.: NOrec: streamlining STM by abolishing ownership records, Vol. 45, No. 5, pp. 67–78 (online), available from ⟨https://academic.microsoft.com/paper/2155500238⟩ (2010).
[9] Felber, P., Fetzer, C. and Riegel, T.: Dynamic Performance Tuning of Word-Based Software Transactional Memory, *Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'08)*, pp. 237–246 (2008).
[10] Dragojevic, A., Guerraoui, R. and Kapalka, M.: Stretching Transactional Memory, *PLDI 2009*, ACM SIGPLAN, (online), available from ⟨https://www.microsoft.com/en-us/research/publication/stretching-transactional-memory/⟩ (2009).
[11] Diegues, N., Romano, P. and Garbatov, S.: Seer: Probabilistic Scheduling for Hardware Transactional Memory, *ACM Trans. Comput. Syst.*, Vol. 35, No. 3, pp. 7:1–7:41 (online), DOI: 10.1145/3132036 (2017).