Vol.2019-ARC-240 No.34
Vol.2019-SLDM-191 No.34
Vol.2019-EMB-53 No.34
2019/2/28

ハードウェア機構の活用による ハイブリッドトランザクショナルメモリ高速化の検討

浅井 優太[1]　山下 淳[1]　小林 龍之介[1]　二間瀬 悠希[1]　塩谷 亮太[2]　五島 正裕[3]　津邑 公暁[1]

概要：トランザクショナルメモリ（TM）のハードウェア実装である HTM は，性能が高いがハードウェア資源の制限により実行できないトランザクションが存在する．一方，TM のソフトウェア実装である STM は，HTM に比べて性能が低いがハードウェア資源の制限を受けないため，HTM で実行できないトランザクションを実行可能である．そこで，HTM でトランザクションを実行できなかった場合，STM にフォールバックしてそのトランザクションを実行するハイブリッドトランザクショナルメモリ（HyTM）が提案されている．HyTM の実装のなかでも HyNOrec は競合検出に使用するメタデータが 1 つであるという点において，性能に優れている．しかし，STM にフォールバックした場合に性能低下が発生するという HyTM が抱える問題は依然として存在する．本論文では，HTM のための機構を HyNOrec の STM を担う NOrec に対して活用することにより高速化できると考え，NOrec に HTM のための機構を活用した手法を提案する．STAMP ベンチマークを用いて評価を行った結果，最大 15.0‘%，平均 4.1%の高速化を確認した．

1. はじめに

マルチコア環境では，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の並列プログラミングモデルが広く利用されている．このプログラミングモデルに基づいたプログラムでは，共有変数に対するアクセスを調停する必要があり，その調停を行う仕組みとして，これまで一般的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よる速度性能の低下や，デッドロックの発生などの問題が生じる可能性がある．さらに，ロックを適切な粒度で使用することは困難であるため，ロックはプログラマにとって必ずしも利用しやすい仕組みではない．

そこで，ロックを代替・補完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）** [1] 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トランザクションの概念をメモリアクセスに適用したものであり，TM を使用する場合は，従来ロックで保護していたクリティカルセクションを含む一連の命令列を，**トランザクション（Tx）**として定義する．そして，共有変数に対するアクセス競合が発生しない限り Tx 同士を投機的に並列実行することで，ロックを用いる場合よりも高い並列性を実現することができる．

なお，TM ではトランザクションの実行が投機的であるため，共有変数の値を更新する際は，更新前と更新後の値を両方保持しておく必要がある．また，Tx を実行するスレッド間において，同一アドレスに対するアクセスが競合しているか否かを監視する必要がある．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）** [2], [3] では，これらのための機構をハードウェアで実現することで，Tx 操作によるオーバヘッドを抑制している．このような利点から，HTM は現在大きな注目を集めており，Intel 社の Haswell [4] 以降のマイクロアーキテクチャや IBM 社のサーバー向けプロセッサ POWER8 [5] などに実装されている．しかし，これらの HTM はハードウェア資源の制限により実行できない Tx が存在し，Tx の完了を保証しない．このような HTM は *best-effort* HTM に分類される．一方，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）** [6] はソフトウェアにより実装されるため Tx 操作によるオーバヘッドが大きく HTM と比較して低速ではあるが，ハードウェア資源の制限を受けないため HTM で実行できない Tx を実行可能である．

そこで，*best-effort* HTM で実行できない Tx が存在した場合，STM にフォールバックすることでその Tx を完了

[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

© 2020 Information Processing Society of Japan

Vol.2019-ARC-240 No.34
Vol.2019-SLDM-191 No.34
Vol.2019-EMB-53 No.34
2019/2/28

させるハイブリッドトランザクショナルメモリ（**Hybrid Transactional Memory**: **HyTM**） [7] が提案されている．HyTM は HTM で Tx を実行する場合は高い性能を発揮できるが，STM にフォールバックした際の性能低下が著しいため，実用性が低い．

STM の実装の中には HyTM に適したものが存在するが，その STM を使用した HyTM においてもフォールバック時に性能低下が発生する問題は依然として存在する．そこで本稿では，この問題点に対して，HyTM では STM にフォールバックした際に HTM 機構が使用されないことに着目し，HTM 機構を活用して STM を高速化することで HyTM の実用性向上を目指す．

2. トランザクショナルメモリ

2.1 トランザクショナルメモリの概要

共有メモリ型プログラミング環境において，ロックを代替・補完す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 Tx の概念をメモリアクセスに適用したものである．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含む一連の命令列をトランザクションとして定義し，これらを投機的に並列実行する．

ロックは，クリティカルセクション同士を排他実行することにより共有変数の一貫性を保証するため，並列性が低いが，TM は Tx 同士を並列実行し，Tx 間のアクセス競合の発生を検知することで共有変数の一貫性を保証するため，ロックと比較して高い並列性を実現することができる．また，TM は Tx を投機的に実行するため，Tx を粗粒度に定義したとしても並列度が損なわれることが少なく，記述性にも優れている．さらにプログラマは，ロックを使用する際に考慮する必要があるデッドロックを意識すること無く Tx を定義できることから，並列プログラムを容易に設計することができる．

TM では共有変数の一貫性を保証するため，並列実行された Tx の実行結果は，それらの Tx を全て逐次実行した場合の結果と等価でなければならない（**Serializability: 直列化可能性**）．これを保証するために，Tx は以下の 3 つの特性を満たす必要がある．

- **Atomicity（不可分性）** Tx はその操作が完全に実行されている，もしくは全く実行されていないのいずれかでなければならない．また，各 Tx 内における処理結果は，トランザクションの終了と同時に観測される．
- **Consistency（一貫性）** Tx の実行結果は，いずれのスレッドが観測した場合も同一でなければならない．
- **Isolation （排他性）** Tx 実行の途中状態は他のスレッドから観測されてはならない．

また，Serializability を Tx 同士についてのみ保証するか，Tx と Tx 外処理との間にも保証するかによって 2 種類のセマンティクスがある [8]．

- **Weak Isolation:** Tx 同士についてのみ Serializability を保証する．
- **Strong Isolation:** Tx 同士に加え，Tx と Tx 外の処理についても Serializability を保証する．

Weak Isolation では，Tx 同士にのみ Serializability を保証するため，Tx を実行中のあるスレッドがアクセスした共有変数の値を，Tx を実行していない他のスレッドが観測する場合がある．このような場合に，プログラマの予期しない無限ループやセグメント違反を引き起こす可能性がある．また，Weak Isolation を保証する TM を使用してプログラムを記述する場合は一般的にアクセスする共有変数を明示する．

一方，Strong Isolation では，Tx 同士および Tx と Tx 外の処理についても Serializability を保証するため，Tx を実行中のあるスレッドがアクセスした共有変数の値を Tx を実行していない他のスレッドが観測することはない．Strong Isolation を保証する TM を用いてプログラムを記述する場合は，アクセスする変数が共有変数であってもプログラム中に明示しないことが一般的である．そして，Tx ないでアクセスするすべての変数を共有変数とみなす．

TM では，以上で述べたセマンティクスに合わせて，Tx の特性を保証する必要があるため，共有変数へのアクセスを監視する．そして，複数トランザクションからの同一アドレスに対するアクセスが確認され，それによってトランザクションの性質が満たされなくなる場合に，この状態を**競合（Conflict）**として検出する．また，これらの操作を**競合検出（Conflict Detection）**という．競合が検出された場合，いずれかのスレッドがトランザクションの途中までの処理結果を破棄する．この操作を**アボート（Abort）**という．これに対し，トランザクション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値の更新を確定する．この操作を**コミット（Commit）**という．なお，トランザクション実行中はトランザクション内で行われた値の更新が確定されるか否かが未定であるため，値が更新された際には，更新前と更新後の値を両方保持しておく必要がある．そこで，TM では一方の値をメモリに，もう一方の値をそのアドレスと共に別領域に保持する．このような値の管理を**バージョン管理（Version Management）**という．また，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前のメモリおよびレジスタの状態を復元し，トランザクションを再実行する．

以上で述べた競合検出機構とバージョン管理機構の実装により，TM は 2 つに大別される．1 つは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**）であり，もう 1 つは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**）であ

© 2020 Information Processing Society of Japan

Vol.2019-ARC-240 No.34
Vol.2019-SLDM-191 No.34
Vol.2019-EMB-53 No.34
2019/2/28

る．HTM は専用ハードウェアにより実装されるため，競合検出やバージョン管理にかかるオーバヘッドを抑制できるため，高い性能を持つ．しかし，HTM はハードウェア資源の制限により，Tx 内でアクセスできる変数の数に限りがあるほか，実行できない命令があるなど，様々な理由で Tx を実行できない場合がある．また，そのような理由から，Tx のコミットが保証されない．このような Tx のコミットを保証しない HTM は *best-effort* HTM と呼ばれる．この *best-effort* HTM では Tx が実行できな場合に備えてプログラマはフォールバックパスを定義する必要がある．Intel 社の Haswell 以降の CPU や IBM 社の POWER8 の CPU で利用可能な HTM も *best-effort* HTM である．

以降，本論文では HTM については現実的な実装である *best-effort* HTM を対象とし，単に HTM と呼ぶ場合，*best-effort* HTM のことを指すこととする．また，HTM は Strong Isolation を保証する実装が多いため，これについても HTM と呼ぶ場合，本論文では Strong Isolation を保証する HTM を指すこととする．

このような HTM に対して，STM はソフトウェアで実装されるため，競合検出やバージョン管理にかかるオーバヘッドが大きく，HTM に比べて性能が低い．しかし，STM は専用ハードウェアを必要とせず，ハードウェア資源の制限を受けないため，HTM で実行できない Tx を実行可能である．STM では競合検出とバージョン管理のためにメタデータを使用するが，それらの定義の仕方や使用方法により，TL2 [9]，tinySTM [10]，NOrec [11] など，多くの実装が存在する．また，STM は比較的バージョン管理と競合検出にかかるオーバヘッドが小さい Weak Isolation を保証する実装が多い．

2.2 ハイブリッドトランザクショナルメモリ

2.1 節で述べたように，HTM にはハードウェア資源の制限により実行できない Tx が存在する．そこで，HTM で Tx を実行できなかった場合，STM にフォールバックしてその Tx を完了させる**ハイブリッドトランザクショナルメモリ（Hybrid Transactional Memory: HyTM）** [7] が提案されている．HTM では Tx を実行できなかった場合，一般的にグローバルロックにフォールバックするため，その間は他のスレッドは Tx を並列に実行することができない．それに対して，HyTM ではグローバルロックではなく STM にフォールバックするため，フォールバックした際にも Tx を並列実行することが可能であり，高い性能が期待できる．

HyTM の研究では一般的に，HTM を用いて実行する Tx を**ハードウェアトランザクション（Hardware-Tx: H-Tx)**，STM を用いて実行する Tx を**ソフトウェアトランザクション（Software-Tx: S-Tx)** と呼ぶ．そして，HyTM には**図 1** に示すように，H-Tx と S-Tx とを排他実行する実装と，並列実行する実装の 2 種類が存在する．なお，図は下向きに時間軸を取っており，それぞれのスレッドで Tx が実行される様子を示している．排他実行する HyTM は図 1(a) のように，ある期間に実行されるすべての Tx を H-Tx として実行するか，S-Tx として実行するかを動的に切り替える HyTM である．

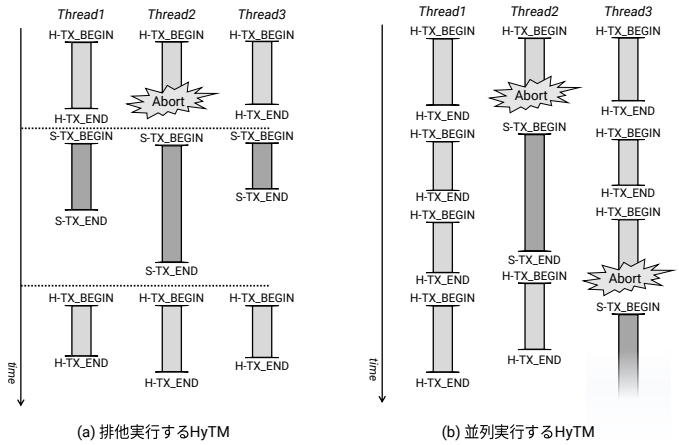

図 1 HyTM の実装方法

ここで，すべての Tx を H-Tx として実行するモードを HW モード，S-Tx として実行するモードを SW モードと呼ぶこととする．排他実行する HyTM はすべての Tx を HW モードで実行するか，SW モードで実行するかを動的に切り替える部分を実装するだけで良いため，実装が容易であるが，モードの切り替え時にすべての Tx をアボートさせるか，すべての Tx の終了を待つ必要があり，切り替えが多発する場合はこの同期にかかるオーバヘッドの影響が大きくなる．また，本来は H-Tx として実行可能である Tx も SW モードでは S-Tx として実行するため，高速な HTM を十分に活かし切れない．さらに，モードの切り替えを行うタイミングの見積もりも容易ではない．

対して，図 1(b) のように H-Tx と S-Tx を並列実行する HyTM はモードの切り替えによるオーバヘッドが発生しないほか，すべての Tx をはじめに H-Tx として実行するため，高速な HTM を十分に活かすことができる．並列実行する HyTM は単に H-Tx と S-Tx を並列に実行するだけでは S-Tx が H-Tx のコミットによる書き込みに起因するアクセス競合を検知することができず，S-Tx を正常に実行することができない．そのため，H-Tx と S-Tx の間の整合性を保つため，H-Tx 側で STM が競合検出に使用するメタデータを更新する必要がある．

STM の実装の中には TL2 のように共有変数ごとにバージョン情報を表すメタデータを持つものがあり，Tx をコミットする際，書き込みを行った共有変数に対するメタデータの全てに対してそのバージョン情報の更新を行う．そのため，このような STM を使用した HyTM では，H-Tx と S-Tx を並列実行する際に H-Tx 内で書き込みを行った

© 2020 Information Processing Society of Japan

Vol.2019-ARC-240 No.34
Vol.2019-SLDM-191 No.34
Vol.2019-EMB-53 No.34
2019/2/28

共有変数に対しても S-Tx と同様にメタデータを更新する必要がある．このとき行われるメタデータの更新は，本来 H-Tx では必要ない処理であるため，H-Tx にオーバヘッドを課すことになる．このように，STM の実装によっては H-Tx 内で更新しなければならないメタデータが多く，H-Tx に多大なオーバヘッドを課してしまう．また，H-Tx 内で多くのメタデータの更新を行うことで，ハードウェア資源を消費してしまい，本来 H-Tx で実行できたはずの Tx を実行できなくなる可能性もある．そのため，更新するべきメタデータが少ない方が HyTM の性能を引き出すことができると考えられる．以上より，高性能な HyTM を実現するためには，H-Tx と S-Tx を並列実行する実装方式であり，その際に H-Tx 内で更新する STM のメタデータが少ないことが必要であると考えられる．

3. NOrec

HyTM に適した STM 実装の一つに **NOrec** [11] がある．NOrec はスレッド間で共有するメタデータとして **global_clock** を持ち，これを使用して Tx 間の整合性を保つ．この global_clock は Tx がコミットされる度にその値が上昇していき，global_clock 自身の値がバージョン情報を表すほか，グローバルロックの役割も持ち，その値が奇数のときロック状態，偶数のときロック解放状態を表す．

また，NOrec はスレッド固有のメタデータとして **Read-Set**，**Write-Set** と呼ばれるバッファを持ち，Tx で読み出した共有変数の値とそのアドレスを Read-Set に，書き込む共有変数の値とそのアドレスを Write-Set に記録する．さらに，読み出した global_clock の値を記録する **snapshot** もスレッド固有のメタデータとして持つ．

次に，NOrec で実行される Tx の動作概要を説明する．Tx を開始するとまず，global_clock の値を snapshot に保持する．そして，Tx 内で読み出した共有変数のアドレスとその値を Read-Set，書き込む共有変数のアドレスとその値を Write-Set に記録する．Tx 終了時には Write-Set に記録された内容を共有メモリに writeback することで Tx をコミットする．writeback 中に他の Tx が共有変数を読み出すのを防ぐために，writeback を行う Tx はまず global_clock をインクリメントし，その値を奇数にすることでグローバルロックを獲得する．その後，writeback を完了すると，再び global_clock をインクリメントし，その値を偶数にすることでグローバルロックの解放とバージョンの更新を行う．writeback を行う際に Read-Set の一貫性が保証されないことが確認された場合，Read-Set の一貫性を検証する **validation** が必要となり，validation により Read-Set の一貫性の保証が確認できた後にグローバルロックの獲得を行う．

なお，先述した Tx 内で行う共有変数の書き込みについては，単に Write-Set に対象の共有変数のアドレスと更新後の値を記録するのみでよいが，共有変数の読み出しについては，validation により Read-Set の一貫性を検証する必要がある．これは，他の Tx の writeback により今までに読みだした共有変数，すなわち Read-Set に記録された共有変数が共有メモリ上で書き換えられることがあり，この状態で共有メモリから新たな共有変数を読み出すと，その後の Tx の実行において更新前の共有変数の値と更新後の共有変数の値を混在して使用してしまう可能性があるからである．

```
1  unsigned Validate(){
2    while(true){
3      time = global_clock;
4      if((time & 1)== 1)
5        contine;
6
7      for each (addr, val) in Read−Set
8        if(∗addr != val)
9          TXAbort();
10
11     if(time == global_clock)
12       return time;
13   }
14 }
```

図 2 Validate 関数

validation では Read-Set に記録された共有変数の値と，共有メモリ上の値を比較することで一貫性を確認する．このとき，いずれかの共有変数の値が共有メモリ上の値を異なっていた場合は Tx をアボートする．validation により Read-Set の一貫性を確認できた場合，Read-Set に読み出し対象の共有変数のアドレスとその値を記録し，共有変数の読み出しを完了する．また，このとき Read-Set の一貫性が確認できた際の global_clock の値を snapshot として保持する．共有変数の読み出し時には，まず global_clock の値が snapshot と等価であるかどうかを判定する．このとき，等価である場合は前回 validation を行ったときからいずれの Tx も writeback を行なっていないことを意味するため，Read-Set の一貫性は保証されており，validation を行う必要はない．しかし，異なっていた場合は前回 validation を行ったときからいずれかの Tx が writeback を行ったことを意味するため，Read-Set の一貫性が保証されないため，validation を行う必要がある．

以上のような NOrec で実行される Tx の validation，共有変数の読み出し，コミットの詳しい動作について，それぞれ擬似コードを用いて説明する．まず，validation については**図 2** に示す Validate 関数で行われる．Validate 関数ではまず，global_clock の値を取得する (line 3)．取得した値が偶数，すなわちグローバルロックが解放されている状態ならば，Read-Set に記録された共有変数のすべてに関して，その値が共有メモリ上の値と異なっていないかをチェックする (line 4-8)．このとき，いずれかの共有変数に

© 2020 Information Processing Society of Japan

Vol.2019-ARC-240 No.34
Vol.2019-SLDM-191 No.34
Vol.2019-EMB-53 No.34
2019/2/28

```
1  Value TXRead(Address addr){
2    if(Write−Set.contains(addr))
3      return Write−Set[addr];
4
5    val = ∗addr;
6    while(snapshot != global_clock){
7      snapshot = Validate();
8      val = ∗addr;
9    }
10
11   Read−Set.append(addr,val);
12   return val;
13 }
```

図 3 TXRead 関数

```
1  void TXCommit(){
2    if(read−only transaction)
3      return;
4
5    while(!CAS(&global_clock, snapshot, snapshot + 1))
6      snapshot = Validate();
7
8    for each (addr, val) in Write−Set
9      ∗addr = val;
10
11   global_clock = snapshot + 2;
12 }
```

図 4 TXCommit 関数

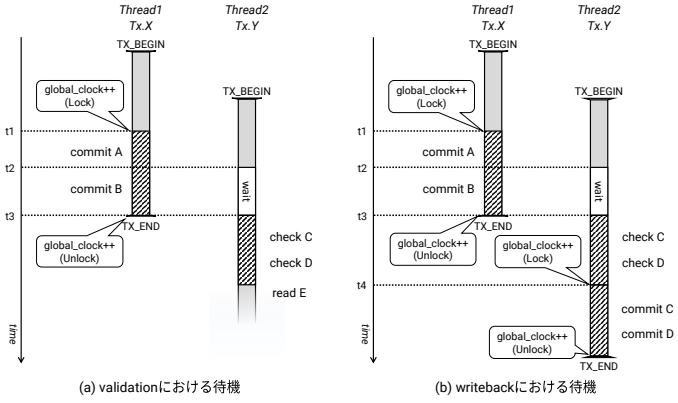

図 5 グローバルロックの解放待機の様子

対して Read-Set に記録されたものと共有メモリ上の値が異なっていた場合，Read-Set に一貫性がないことが確認されるため，Tx をアボートする (line 9)．最後に，3 行目で読み出した global_clock が，Read-Set の一貫性検証中に更新されていないことを確認し，その値を返す (line 11-12)．このとき，global_clock が更新されていた場合，Read-Set の検証中に他の Tx が writeback を行ったことを意味する．そのため，一貫性検証のために読み出した共有変数が他の Tx の writeback により書き換えられた可能性がある．このような理由から正常に Read-Set の一貫性検証ができなかったため，3 行目に戻り，validation を始めからやり直す必要がある．また，4 行目で global_clock の値が奇数である，すなわちグローバルロックが獲得されていた場合も他の Tx の writeback により一貫性検証のために読み出した共有変数が書き換えられる可能性がある．そのため，グローバルロックが解放されるまで待機する必要がある．

次に，Tx 内で共有変数を読み出す際の動作は**図 3** に示す TXRead 関数で行われる．TXRead 関数を開始するとまず，対象がすでに Write-Set に記録されている場合，Write-Set から読み出す (line 2-3)．読み出し対象が Write-Set に記録されていない場合は共有メモリから読み出しを行う (line 5)．次に，global_clock と snapshot の値を比較し (line 6)，等しい場合は読みだした共有変数のアドレスとその値とを Read-Set に記録し，共有変数の読み出しを完了する (line 11-12)．それに対し，global_clock と snapshot の値を比較し，異なっている場合，他の Tx の writeback により現在までに読みだした共有変数の値，すなわち Read-Set に記録されている共有変数の値が書き換えられた可能性があるため，Read-Set の一貫性を検証する，**validation** を行う必要がある (line 7)．

最後に，Write-Set の内容を共有メモリに writeback し，Tx のコミットを行う TXCommit 関数ついて**図 4** を用いて説明する．まず，TXCommit 関数を開始すると，共有変数の読み出しのみを行う Tx に関しては最後の共有変数の読み出し時に Read-Set の一貫性が保証されていることを確認できているためそのままコミットを完了する (line 2-3)．対して，writeback を行う Tx は snapshot と global_clock の値が等価である，すなわち Read-Set の一貫性が保証されているとき，writeback 中に他の Tx が共有変数から値を読み出すことを防ぐために global_clock をインクリメントし，その値を奇数にすることでグローバルロックを獲得する (line 5)．グローバルロックの獲得に成功した場合，Write-Set の内容を共有変数に writeback する (line 8-9)．writeback を完了すると，global_clock に snapshot + 2 を代入し，その値を偶数にすることでグローバルロックを解放し，コミット処理を完了する (line 11)．なお，グローバルロックの獲得に失敗した場合は他の Tx が writeback を行っており，これにより Read-Set の一貫性が保証されない状態となる．その場合は validation を行い，Read-Set の一貫性を検証する必要がある (line 6)．

以上に述べたように，NOrec で実行される Tx では，validation，writeback 時に他スレッドによってグローバルロックが獲得されていた場合，それが解放されるまで待機する必要がある．これらの待機時間について，2 つの Tx が並列実行している様子を表した**図 5** を用いて説明する．まず，validation 時に発生する待機時間を図 5(a) を例にして述べる．図 5 は下向きに時間軸を取っており，*Thread*1 と *Thread*2 がそれぞれ *Tx.*X，*Tx.*Y を実行している様子を表している．また，図中の commit X は Write-Set に記録さ

© 2020 Information Processing Society of Japan

Vol.2019-ARC-240 No.34
Vol.2019-SLDM-191 No.34
Vol.2019-EMB-53 No.34
2019/2/28

れた共有変数 X についての writeback を表し，check X は Read-Set に記録された共有変数 X について validation を行うための X に対する読み出しを表す．*Thread*1 の *Tx.*X は writeback を行うために，global_clock をインクリメントし，グローバルロックを獲得する (t1)．その後，*Thread*2 の *Tx.*Y の実行が進み，共有変数 E の読み出しを試みるとする (t2)．このとき，自身が持つ snapshot の値が global_clock の値と異なるため，validation を試みるが，*Thread*1 の *Tx.*X がグローバルロックを獲得済みであるため，解放されるまで待機する．その後，*Thread*1 の *Tx.*X が writeback を完了し，global_clock の値を再度インクリメントしてグローバルロックを解放すると，*Thread*2 の *Tx.*Y は validation を行うことが可能となる (t3)．validation を完了した *Tx.*Y は共有変数 E を読み出し，*Tx.*Y の実行を進める．このように，いずれかの Tx がグローバルロックを獲得している間，他の Tx は validation を行うことはできず，グローバルロックの解放を待機する．

次に，writeback 時に発生する待機時間について図 5(b) を用いて述べる．図 5(b) では先と同様に，*Thread*1 と *Thread*2 がそれぞれ *Tx.*X，*Tx.*Y を実行している様子を表している．まず，*Thread*1 の *Tx.*X は writeback を行うために，global_clock をインクリメントし，グローバルロックを獲得する (t1)．ここで，*Thread*2 の *Tx.*Y も writeback を試みるが，global_clock の値が奇数であり，グローバルロックが獲得されている状態のため，global_clock を獲得することはできない (t2)．また，このとき *Thread*2 の *Tx.*Y は validation が必要となるが，グローバルロックが解放されるまで待機する．その後，*Thread*1 の *Tx.*X がグローバルロックを解放すると，*Thread*2 の *Tx.*Y は validation を行うことが可能となるため，snapshot を最新の global_clock の値に更新する (t3)．そして，*Thread*2 の *Tx.*Y は 2 度目のグローバルロック獲得を試みるが，この場合は snapshot と global_clock の値が等価であるため，グローバルロックの獲得に成功し，writeback を行うことが可能となる (t4)．このように，いずれかの Tx がグローバルロックを獲得している間，他の Tx は writeback を行うことができず，グローバルロックの解放を待機する．

以上で示したように NOrec において実行される Tx にはグローバルロックの待機が存在し，writeback を行っている Tx が存在する場合は他の Tx は validation や writeback を行うことができない．これらの待機時間は NOrec の性能低下の要因と考えられ，待機時間を削減することで NOrec の性能を改善することができると考えられる．

以上で説明した NOrec の HyTM 実装である HyNOrec [12] が提案されている．NOrec はスレッド間で共有するメタデータが global_clock のみであるため，HyNOrec における H-Tx で更新する必要があるメタデータも global_clock のみでよい．2.2 節でも述べたように，HyTM で実行される H-Tx は更新するメタデータが少ないほうが高い性能を発揮できるため，NOrec は HyTM に適した STM であり，HyNOrec は HyTM の中でも高い性能を発揮すると考えられる．しかし，HyNOrec においても，S-Tx にフォールバックした際に，性能低下が起きる HyTM が抱える問題は依然として存在する．そこで，本論文では HyNOrec の高速化のために，NOrec の高速化を図る．

表 1 調査環境

OS	Ubuntu 16.04
Processor	Intel Xeon Gold 6152
clock	2.10 GHz
physical/logical #cores	22/44 cores
L1-dcache	32 KBytes
L2-cache	1024 KBytes
L3-cache	30976 KBytes
Memory	16 GBytes
Compiler	gcc version 5.5.0
Compile options	-O3

表 2 ベンチマークパラメタ

Genome	-g16384 -s64 -n16777216
Intruder	-a10 -l128 -n262144 -s1
Kmeans-High	-m15 -n15 -t0.00001 -i random-n65536-d32-c16.txt
Kmeans-Low	-m40 -n40 -t0.00001 -i random-n65536-d32-c16.txt
Labyrinth	-i random-x512-y512-z7-n512.txt
Ssca2	-s20 -i1.0 -u1.0 -l3 -p3
Vacation-High	-n4 -q60 -u90 -r1048576 -t4194304
Vacation-Low	-n2 -q90 -u98 -r1048576 -t4194304
Yada	-a15 -i ttimeu1000000.2

4. NOrec の性能調査

4.1 待機時間の調査

NOrec のグローバルロックに起因する待機時間が，Tx 内の処理でどれほどの割合を占めるかを調査した．なお，writeback の待機時間に関しては，グローバルロックの獲得に失敗したときから，グローバルロックの獲得に成功するまでの時間を writeback を待機したとみなし，これを待機時間とした．このとき，グローバルロックの獲得に成功するまでに validation を行うため，validation の待機時間に関しては，共有変数読み出し時に行う validation のみを対象とした．

4.1.1 調査環境

調査は 16 スレッドで行った．また，調査環境を**表 1** に示す．調査に用いたプログラムは TM の研究に広く用いられている STMAP [13] から，Genome，Intruder，Kmeans-High，Kmeans-Low，Labyrinth，SSCA2，Vacation-High，Vacation-Low，Yada の 7 つを選択した．また，実行に使用した入力パラメタを**表 2** に示す．

© 2020 Information Processing Society of Japan

Vol.2019-ARC-240 No.34
Vol.2019-SLDM-191 No.34
Vol.2019-EMB-53 No.34
2019/2/28

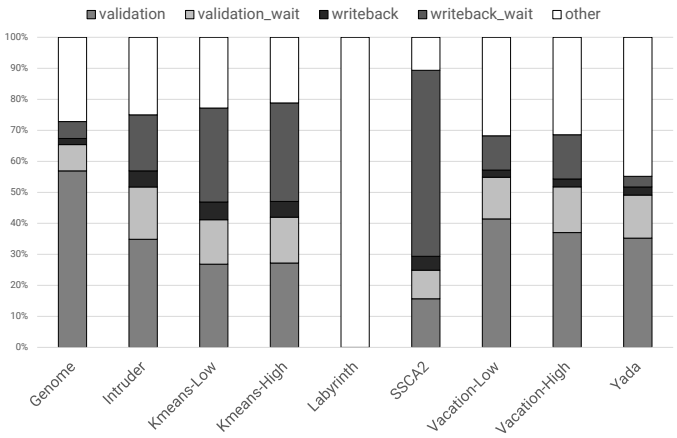

図 6 Tx 内の処理の内訳

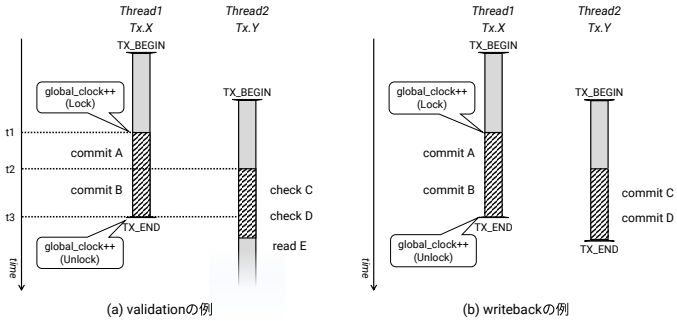

図 7 グローバルロックの解放を待機する必要がない例

4.1.2 調査結果

調査結果を**図 6** に示す．図中の凡例はそれぞれ以下の通りである．

validation: validation に要した時間
validation_wait: validation における待機時間
writeback: writeback に要した時間
writeback_wait: writeback における待機時間
other: 上記以外の処理に要した時間

調査結果より，いくつかのプログラムで待機時間が顕著であり，validation における待機時間に関しては平均 11.8 %，最大で 16.7 %，writeback における待機時間に関しては平均 19.4 %，最大 60.0 %であることが確認できた．

4.2 待機時間に関する考察

グローバルロックによる待機時間は Tx 間の整合性を保つために必要であるが，**図 7** のように，validation と writeback のそれぞれにおいて異なる Tx 間で同一共有変数へのアクセスが発生しない場合，グローバルロックの待機が必要ない．図 7(a) のグローバルロックの解放を待機する必要がない例では，*Thread*1 の *Tx.*X が writeback を，*Thread*2 の *Tx.*Y が validation を行っているが，writeback 対象の共有変数と validation 対象の共有変数が異なる．この場合，*Thread*2 がグローバルロックの解放を待機せずに validation を行っても，2 つの Tx 間で同一共有変数に対するアクセスが存在しないため，*Thread*2 の *Tx.*Y は *Tx.*X の writeback と並列に validation を行うことが可能である．しかし，既存の NOrec では同一共有変数へのアクセスが発生する場合とそうでない場合を区別できないため．また，図 7(b) のように異なる Tx が並列に writeback を行う場合についても同様に，同一共有変数へのアクセスが発生していないにもかかわらず，NOrec ではグローバルロックの待機を行う必要がある．

以上のように，NOrec では実際にはグローバルロックの解放を待機することなく，validation，writeback を並列に行うことができる共有変数へのアクセスパターンが存在する．これらのパターンについてグローバルロックの解放を待機することなく validation，writeback を行うことを可能とすることで待機時間を削減し，NOrec の高速化を実現することができると考えられる．

5. HTM を活用した NOrec

HyTM では，H-Tx から S-Tx にフォールバックした際，H-Tx 実行時に使用されていた HTM 機構は通常使用されない．そこで S-Tx の実行に用いる NOrec に HTM 機構を活用することで 4 章で示した待機時間を削減し，高速化を図る．

5.1 HTM を活用した validation

4.2 節で示したように，NOrec には validation と writeback とを並列実行できる共有変数へのアクセスパターンが存在するが，同一共有変数へのアクセスを検知することができないため，validation を行う Tx はグローバルロックの解放を待機する必要があった．そこで，HTM を活用し，validation を H-Tx として定義することで，グローバルロックの解放を待機する間にも validation を投機的に実行する手法を提案する．提案手法では，H-Tx 内で validation を行うため，writeback 中の共有変数とのアクセス競合を検知できるようになり，グローバルロックの解放を待機する必要なく validation を行うことが可能である．また，提案手法では HTM が Strong Isolation を保証するため，H-Tx で実行する validation と H-Tx 外で実行する writeback の同一共有変数へのアクセス競合が発生した場合 validation を行う H-Tx をアボートする．以降は NOrec で実行する Tx を S-Tx と呼ぶことにし，H-Tx と区別する．

HTM を活用した投機的な validation を実現するために，Validate 関数のグローバルロックを待機する部分に変更を加え，**図 8** のようにした．ここで，この疑似コードを用いて投機的な validation について説明する．なお，図中の HTX_BEGIN は H-Tx の開始を表し，HTX_END は H-Tx のコミットを意味する．ABORT ラベルは H-Tx が競合などの理由でアボートした際のジャンプ先を表し，HTX_ABORT は明示的に H-Tx をアボートさせることを意味する．また，RETRY ラベルについては，H-Tx がア

© 2020 Information Processing Society of Japan

Vol.2019-ARC-240 No.34
Vol.2019-SLDM-191 No.34
Vol.2019-EMB-53 No.34
2019/2/28

```
1  unsigned Validate(){
2    while(true){
3      time = global_clock;
4      if((time & 1)== 1){
5  RETRY:
6        HTX_BEGIN();
7        for each (addr, val) in Read−Set
8          if(∗addr != val){
9            HTX_END();
10           STXAbort();
11         }
12
13       time = global_clock;
14       if((time & 1) == 1)
15         HTX_ABORT();
16
17       HTX_END();
18       return time;
19 ABORT:
20       if(htx_retry)
21         goto RETRY;
22       contine;
23     }
24
25     for each (addr, val) in Read−Set
26       if(∗addr != val)
27         STXAbort();
28
29     if(time == global_clock)
30       return time;
31   }
32 }
```

図 8 HTM を活用した Validate 関数

ボートした場合に再度 H-Tx を実行する際のジャンプ先として使用し，変数 htm_retry については，H-Tx をリトライをする場合 true であるような変数とする．なお，リトライを行う場合の条件やその回数については 6 章で述べる．そして，NOrec で実行する Tx を S-Tx と呼ぶことにしたため，図 2 で示した Validate 関数の TXAbort() に関しても STXAbort() とし，H-Tx のアボートと区別する．

スレッドが Validate 関数を開始した場合，通常 NOrec と同様にまずグローバルロックを確認する．グローバルロックが他のスレッドにより獲得されていた場合，提案手法では HTX_BEGIN により H-Tx を開始する (line 6)．その後，Read-Set に記録されたすべての共有変数に関して，共有メモリ上の値と異なっていないかを確認する (line 7-8)．このとき，writeback している S-Tx が H-Tx 内ですでに読み出した共有変数に対する書き込みを行った場合，H-Tx は競合を検知し，アボートする．そのため，一貫性のない共有変数の値の読み出しを防ぐことができる．また，Read-Set に記録された共有変数のいずれかが共有メモリ上の値と異なっていた場合は，HTX_END で H-Tx をコミットしてから自身の S-Tx をアボートする (line 9-10)．Read-Set に一貫性があることが確認できた場合，global_clock の値を読み出し，その値が奇数でない，つまりグローバルロックが獲得されていないかを確認し，もしグローバルロックが獲得されていた場合，H-Tx をアボートする (line 13-15)．これは，他の S-Tx の writeback 中に validation を完了させてしまうと，共有メモリに一貫性がない状態で validation を完了させてしまう可能性があるためである．最後に，HTX_END で H-Tx をコミットし，3 行目で読み出した global_clock の値を返す (line 17-18)．なお，H-Tx がアボートした際は，19 行目にジャンプする．その後 htm_retry が true であり，H-Tx のリトライを行う場合は，5 行目に戻り，再度投機的な validation を行う．リトライを行わない場合は 3 行目に戻り，validation を始めからやり直す．このときグローバルロックが獲得されている場合は再度投機的な validation を行い，グローバルロックが獲得されていなかった場合はグローバルロックの解放を待機する必要がないため，既存の NOrec と同様の validation を行う．

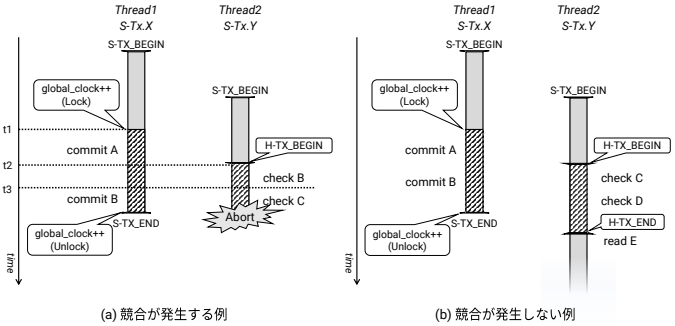

図 9 HTM を活用した validation の様子

以上のような投機的な validation が行われる様子を複数の S-Tx が並列実行している**図 9** を例にして説明する．はじめに，図 9(a) の同一共有変数へのアクセスが発生する場合について説明する．まず，*Thread*1 の *S-Tx.*X がグローバルロックを獲得し，writeback を開始する (t1)．次に，*Thread*2 の *S-Tx.*Y が共有変数の読み出しの際に validation を必要としたとする．このとき，グローバルロックが他の S-Tx により獲得されているため，HTM を用いた投機的な validation を開始する (t2)．その後，*S-Tx.*Y が時刻 t3 の前に Read-Set の一貫性検証のために共有変数 B の値を読み出す．そして，時刻 t3 の後に *Tx.*X の writeback により，共有変数 B の値が更新されたとする．このとき，HTM が *S-Tx.*Y の H-Tx 内で読みだした共有変数 B との競合を検知し，H-Tx をアボートすることで一貫性のない共有変数の読み出しを防ぐ．対して，図 9(b) のように同一共有変数へのアクセスが発生しない場合は，同一共有変数へのアクセス競合が発生しないため，*Thread*2 は H-Tx をコミットし，validation を完了することができる．

以上のように，提案手法では writeback 対象の共有変数が validation 対象であった場合でも HTM により競合を検知し，一貫性のない共有変数の読み出しを防ぐことが

© 2020 Information Processing Society of Japan

Vol.2019-ARC-240 No.34
Vol.2019-SLDM-191 No.34
Vol.2019-EMB-53 No.34
2019/2/28

```
1  void TXCommit(){
2    if(read−only transaction)
3      return;
4
5  RETRY:
6    HTX_BEGIN()
7    for each (addr, val) in Read−Set
8      if(∗addr != val)
9        HTX_END();
10       STXAbort();
11
12   for each (addr, val) in Write−Set
13     ∗addr = val;
14
15   if((global_clock & 1) == 1)
16     HTX_ABORT();
17
18   global_clock += 2;
19   HTX_END();
20   return;
21
22 ABORT:
23   if(retry)
24     goto RETRY;
25
26   while(!CAS(&global_clock, snapshot, snapshot + 1))
27     snapshot = Validate();
28
29   for each (addr, val) in Write−Set
30     ∗addr = val;
31
32   global_clock = snapshot + 2;
33 }
```

図 10 HTM を活用したコミット関数

できる．そのためグローバルロックの解放を待機せずに validation を行うことが可能であり，競合が発生しない場合は，そのまま validation を完了することができる．

5.2 HTM を活用した writeback

本節では 5.1 節で示した待機時間削減手法とは別の手法として，NOrec の writeback に HTM を活用したグローバルロックの獲得と解放とを用いない writeback を投機的に実行する手法を提案する．提案手法では，H-Tx で writeback を行うため，writeback を並列実行しても同一共有変数への書き込みを検知できるようになり，グローバルロックによる待機を行わずに並列に writeback を行うことが可能である．なお，writeback のみを並列実行する場合，他の S-Tx の writeback により Read-Set の一貫性が失われる可能性があるため，H-Tx 内で validation も行うようにし，Read-Set の検証中に読みだした共有変数についても競合を検知できるようにする．また，global_clock のインクリメントによるグローバルロックの獲得と解放とを行わないため，H-Tx の最後でロック状態を伴わずに global_clock のバージョン情報を更新するようにする．

HTM を活用した投機的な writeback を実現するために，

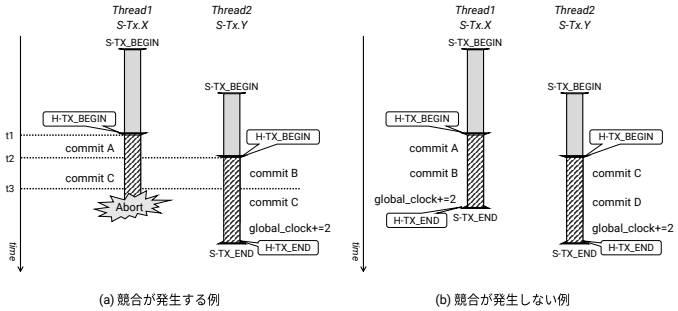

図 11 HTM を活用した writeback の様子

TXCommit 関数のグローバルロックを獲得して writeback を行う部分に変更を加え，**図 10** のようにした．ここで，この疑似コードを用いて提案手法による writeback について説明する．なお，図中の HTX_BEGIN，HTX_END，HTX_ABORT，ABROT ラベル，RETRY ラベル，変数 retry については，5.1 節で説明したとおりである．提案手法では，TXCommit 関数を開始した S-Tx はまず H-Tx を開始する (line 6)．その後，Read-Set の一貫性を確認する (line 7-10)．Read-Set の一貫性が確認できた場合，Write-Set の内容を共有メモリに writeback する (line 12-13)．次に global_clock の値が奇数でない，すなわちグローバルロックが獲得されていないことを確認し，もしグローバルロックが獲得されていた場合は H-Tx をアボートする (line 15-16)．これは，他の S-Tx の writeback 中に H-Tx をコミットし，H-Tx で行った writeback の内容を共有メモリに反映すると，共有メモリの一貫性が失われる可能性があり，それを防ぐためである．最後に，global_clock の値に 2 を加算し，H-Tx をコミットすることで TXCommit 関数を完了する．(line 18-20)．なお，H-Tx がアボートした際は，H-Tx をリトライするかを判定し，リトライする場合については 5 行目に戻り，H-Tx をやり直す (line 22-24)．リトライを行わない，すなわち H-Tx が競合の多発やハードウェア資源の制限によりコミットできなかった場合，26 行目以降の既存の NOrec と同様の writeback を行う．

以上のような投機的な writeback が行われる様子を複数の S-Tx が並列実行している**図 11** を用いて説明する．はじめに，図 11(a) のように並列に実行される 2 つの S-Tx 間において同一共有変数への writeback が発生する場合について考える．なお，提案手法では H-Tx を開始後にまず validation を行うが，この例では S-Tx 内で共有変数への書き込みのみを行う validation が必要ない例を示す．まず，*Thread*1 の *S-Tx.*X が提案手法による HTM を用いた投機的な writeback を開始する (t1)．その後，*Thread*2 の *S-Tx.*Y も提案手法による HTM を用いた投機的な writeback を開始する (t2)．そして，*S-Tx.*X が時刻 t3 の前に *S-Tx.*X が共有変数 C を writeback し，*S-Tx.*Y が時刻 t3 の後に共有変数 C を writeback したとする．このとき，HTM が同一共有変数への書き込みを検知するため，一方の H-Tx をアボー

© 2020 Information Processing Society of Japan

Vol.2019-ARC-240 No.34
Vol.2019-SLDM-191 No.34
Vol.2019-EMB-53 No.34
2019/2/28

トすることで同一共有変数への書き込みを防ぐ．図 11(a) の例では，*Thread*1 の H-Tx をアボートすることで同一共有変数への書き込みを防いでいる．また，図 11(a) の例では示されていないが，H-Tx 内の最初に validation を行っているため，Read-Set に記録された共有変数に対して他の S-Tx が書き込みを行ったことを検知できる．そのため，Read-Set の一貫性が失われた場合に誤って writeback してしまうことはない．対して，図 11(b) のように，同一共有変数へのアクセスが発生しない場合は，競合が発生しないため，*S-Tx.*X と *S-Tx.*Y の writeback が並列実行できる．なお，global_clock の更新は H-Tx の最後に行うため，*S-Tx.*X の H-Tx が完了したのちに *S-Tx.*Y の H-Tx が global_clock の更新を行う．そのため，この例では global_clock による競合は発生しない．

以上のように，提案手法では，異なる S-Tx で writeback 対象の共有変数が同一である場合も HTM により競合を検知し，同一共有変数への書き込みを防ぐことができる．そのためグローバルロックの解放を必要とせずに投機的に writeback することが可能であり，競合が発生しない場合はそのまま writeback を完了することができる．

6. 評価結果

6.1 評価環境

評価には第 4 世代以降の Intel プロセッサに搭載されている HTM である RTM を使用し，16 スレッドで行った．また，評価環境は 4 章の予備調査と同様であるが，提案手法はコンパイラオプションとして-O3 の他に-mrtm を使用した．

6.2 リトライポリシー

H-Tx がアボートした際のリトライについて，HTM を活用した validation ではグローバルロックが獲得されていた際の明示的なアボートが行われた場合のみリトライを行うこととした．これはグローバルロックが獲得されていることによって明示的にアボートしたならば，次に global_clock の値を確認した際も global_clock の値が奇数であり，結局は再び投機的な validation を行うためである．

HTM を活用した writeback については，リトライを設けなかった．これは評価に使用したプログラムにおいて，リトライを設けなかった場合が全体として最も良い結果を示したからである．

6.3 validation の評価

HTM を用いた投機的な validation を適用した NOrec の評価結果を**図 12** に示す．図 12 では，各プログラムについて実行時間を 2 本のバーで示している．これらのバーは左側が既存の NOrec，右側が提案手法を適用した NOrec を表しており，既存の NOrec を 1 として正規化してある．また，**表 3** に投機的な validation を行った際，それが成功した割合を示す．

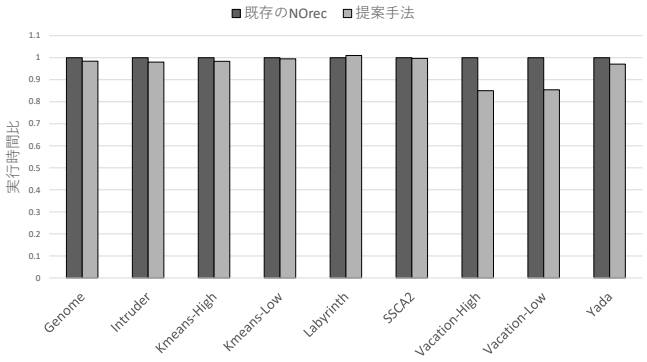

図 12 validation の実行時間比

表 3 投機的な validation の成功率

Genome	18.8%
Intruder	29.2%
Kmeans-High	22.7%
Kmeans-Low	25.7%
Labyrinth	11.4%
SSCA2	4.2%
Vacation-High	83.9%
Vacation-Low	79.8%
Yada	53.2%

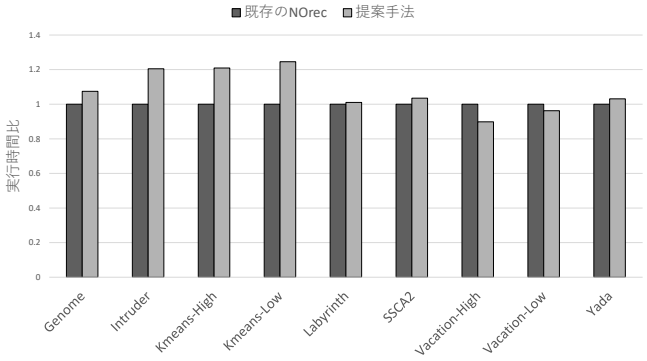

図 13 writeback の実行時間比

結果より，すべてのプログラムについて既存の NOrec と同等か，それ以下の実行時間を達成していることが確認でき，平均 4.1%の実行時間を削減を達成した．特に，Vacation-High，Vacation-Low が実行時間を 15.0%と大きく削減することができた．また，表 3 より Vacation-High，Vacation-Low は投機的な validation の成功率が最も高かったことも確認できる．

6.4 writeback の評価

HTM を用いた投機的な writeback を適用した NOrec の評価結果を**図 13** に示す．図 13 は，図 12 と同様に 2 本のバーの内，左側が既存の NOrec，右側が提案手法を適用した NOrec を表しており，既存の NOrec を 1 として正規化してある．

© 2020 Information Processing Society of Japan

Vol.2019-ARC-240 No.34
Vol.2019-SLDM-191 No.34
Vol.2019-EMB-53 No.34
2019/2/28

表 4　投機的な writeback の成功率

Genome	60.3%
Intruder	4.5%
Kmeans-High	0.4%
Kmeans-Low	1.0%
Labyrinth	51.3%
SSCA2	0.5%
Vacation-High	33.4%
Vacation-Low	27.6%
Yada	15.0%

また，**表 4** に投機的な writeback を行った際，それが成功した割合を示す．

結果より，ほとんどのプログラムにおいて既存の NOrec よりも性能が悪化していることが確認できる．この要因として，global_clock 自体の更新による競合の多発が考えられる．投機的な writeback を行う際，global_clock の更新タイミングが重なることがある．このとき，一方の H-Tx をアボートさせる必要があり，このような競合が多発したことため，投機的な writeback を行う提案手法では既存の NOrec に比べて性能が悪化したと考えられる．

7. おわりに

本論文では，HyTM に適した STM である NOrec の validation と writeback のそれぞれについてグローバルロックの解放を待機する必要がない共有変数へのアクセスパターンが存在することを確認した．また，HTM を活用することで，グローバルロックの解放の待機を必要とせず，同一共有変数へのアクセスが発生しない場合に処理を完了することができる投機的な validation と writeback を提案した．

提案手法の有効性を示すため，TM の研究に広く用いられている STAMP で評価を行った．評価の結果，HTM を活用した投機的な validation については最大 15.0 %，平均 4.1 %の実行時間削減が確認できた．一方，HTM を活用した投機的な writeback については最大 10.2 %の実行時間を削減できたが，多くのプログラムで実行時間の削減ができなかった．また，この要因として global_clock の競合の多発がその 1 つであることが考えられる．

今後の課題として，投機的な writeback における global_clock の競合を回避するような global_clock の適切な更新方法について検討することが挙げられる．また，今後の展望として，提案手法を適用した NOrec を STM として用いた高速な HyTM の実現を目指す．

謝辞　本研究の一部は，JSPS 科研費 JP17H01711，JP17H01764，JP17K19971 の助成を受けたものである．

参考文献

[1] Herlihy, M. et al.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. 20th Int'l Symp. on Computer Architecture (ISCA'93)*, pp. 289–300 (1993).

[2] Knight, T.: An Architecture for Mostly Functional Languages, *Proc. ACM Conference on LISP and Functional Programming (LFP'86)*, pp. 105–112 (1986).

[3] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C. and Olukotun, K.: Transactional Memory Coherence and Consistency, *Proc. 31st Annual Int'l Symp. Computer Architecture (ISCA'04)*, pp. 102–113 (2004).

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

[5] International Business Machines Corporation: Power ISA® Version 2.07, `https://www.power.org/documentation/power-isa-version-2-07/` (2013).

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

[7] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M. and Nussbaum, D.: Hybrid transactional memory, *Proc. 12th Int'l Conf. on Architectural support for programming languages and operating systems (ASPLOS'06)*, pp. 336–346 (2006).

[8] Dalessandro, L. and Scott, M. L.: Strong isolation is a weak idea, *In TRANSACT'09: 4th Workshop on Transactional Computing* (2009).

[9] Dice, D., Shalev, O. and Shavit, N.: Transaction Locking II, *Proc. 20th Int'l Conf. on Distributed Computing (DISC'06)*, pp. 194–208 (2006).

[10] 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).

[11] Dalessandro, L., Spear, M. F. and Scott, M. L.: NOrec: streamlining STM by abolishing ownership records, *Proc. 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'10)*, pp. 67–78 (2010).

[12] Dalessandro, L., Carouge, F., White, S., Lev, Y., Moir, M., Scott, M. L. and Spear, M. F.: Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory, *ACM SIGARCH Computer Architecture News*, Vol. 39, No. 1, pp. 39–52 (2011).

[13] 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).

© 2020 Information Processing Society of Japan