Vol.2018-ARC-232 No.7
2018/7/30

軽量な一貫性検証によるソフトウェアトランザクショナルメモリの並行性向上に関する検討

飯田 凌大[1]　津邑 公暁[1]

概要：トランザクショナルメモリ（TM）は，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，これを投機的に並列実行することで，粗粒度ロックと同程度の記述性と，細粒度ロックと同等以上の性能とを両立しうるパラダイムとして期待されている．この TM をソフトウェア上に実装したソフトウェアトランザクショナルメモリ（STM）では一般に，論理タイムスタンプを用いて共有変数の一貫性を検証しているが，この手法はスケーラビリティに欠けるという問題がある．また，他の一貫性検証手法として共有変数別ごとのバージョン情報を用いるものがある．この手法は論理タイムスタンプを用いる手法に比べスケーラビリティに優れるが，トランザクション内で読み出す共有変数の個数の二乗に比例するオーバヘッドが発生するという問題がある．本稿では，実行時のスレッド数とトランザクション内で読み出す共有変数の個数とを基準とし，一貫性検証手法を適切なものへ動的に切り替えることで，STM のオーバヘッド抑制とスケーラビリティ向上との両立を目指す．提案手法を実装し，評価を行った結果，最大 27.0%，平均 15.3%の速度向上を達成した．

1. はじめに

マルチコアプロセッサの普及に伴い，プログラマが容易に並列処理を記述できる，共有メモリ型並列プログラミングの重要性が増している．この共有メモリ型並列プログラミングでは，共有変数へのアクセスを調停するために一般的にロックが用いられることが多いが，クリティカルセクションの排他実行に伴う並列度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の粒度を設定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ものではない．

そこで，プログラマビリティが高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 とは，データベースにおけ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，従来ロックを用いてクリティカルセクションとして保護していた区間を**トランザクション（Transaction: Tx）**として定義し，これを投機的に並行実行することで，粗粒度ロックと同程度の記述性と，細粒度ロックと同等以上の性能とを両立しうると期待されているパラダイムである．なお，Tx 実行中は，その実行が投機的であるため，共有変数に対する更新の際には更新前の値と更新後の値の両方を保持しておく必要がある．また，Tx を実行するスレッド間で，共有変数に対する一貫性が損なわれていないかを検査する必要がある．

これらの処理をソフトウェア上に実装した**ソフトウェアトランザクショナルメモリ（Software Transactional Memory: STM）**[2] では一般に，論理タイムスタンプを用いて共有変数の一貫性を検証している [3][4]．この手法では，この論理タイムスタンプ自体に対するアクセスがボトルネックとなり，スケーラビリティに欠けるという問題がある．また，他の一貫性検証手法として共有変数ごとのバージョン情報を用いるものがある [5][6]．この手法は論理タイムスタンプを用いる手法に比べスケーラビリティに優れるが，Tx 内で読み出す共有変数の個数の二乗に比例するオーバヘッドが発生するという問題がある．本稿では，実行時のスレッド数と Tx 内で読み出す共有変数の個数とを基準として，一貫性検証手法を適切なものへ動的に切り替えることで，STM のオーバヘッド抑制とスケーラビリティ向上との両立を目指す．

2. STM とその問題

本章ではまず，STM の概要について述べる．次に，STM における一貫性検証手法とその問題について述べる．

[1] 名古屋工業大学
Nagoya Institute of Technology

2.1 STM

STM における Tx は以下の 2 つの性質を保証する．

Serializability（直列化可能性）：

並行実行された Tx の実行結果は，当該 Tx をある順序で逐次実行した場合と同じであり，すべ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）：

Tx 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はでなければならず，各 Tx 内における処理は Tx の終了と同時に観測される．そのため，処理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，STM では，共有変数の読み出し時および Tx 終了時に，Tx 内で読み出した共有変数の一貫性を検証する．ここで，ある Tx 内でアクセスされたメモリアドレスと他の Tx 内でアクセスされたメモリアドレスとが同一であった場合，上で述べた Tx の性質を満たさなくなることがある．この状態を**競合**として検出する．また，Tx の性質を保証するために行うこれらの操作を**一貫性検証**という．競合が検出された場合，STM では一方のスレッドが Tx の実行を破棄することで競合を回避する．この操作を**アボート**という．その後，Tx をアボートしたスレッドは，メモリの状態を Tx 開始時点の状態に復元し，処理を再実行する．これに対し，競合が検出されずに Tx 処理が完了した場合，Tx 内で行った共有変数に対する値の更新を確定する．この操作を**コミット**という．なお，Tx を投機実行する場合，Tx がアボートされ，途中までの実行結果が破棄される可能性があるため，Tx 内で共有変数に対する更新の際には，更新前の値と更新後の値の両方を保持しておく必要がある．そのため，STM では更新後の値をそのアドレスとともにスレッドローカルな領域に記録し，Tx のコミット時にその値をグローバルな領域に上書きする．このようなデータ管理を**バージョン管理**という．

2.2 Opacity

前節で述べた Tx が満たすべき 2 つの性質の他に，STM が保証すべき性質として **Opacity**[7][8] がある．Opacity とは，実行中の Tx は（たとえそのスレッドがアボートするとしても）ある順序で逐次実行している場合と等価でなければならない，という性質である．つまり Opacity は，Tx 内で読み出した共有変数すべての値はいずれかの時点におけるスナップショットと等価である，という性質と言い換えることができる．この性質が保証されていない場合，ある Tx は他の Tx が更新する前の値と更新した後の値とを混在させて読み出す可能性があり，これはプログラマが予期しない無限ループやセグメント違反を引き起こす場合がある．

時刻	*Thread*1	*Thread*2
t0	A=0; B=0;	
t1	TX_BEGIN(0);	TX_BEGIN(1);
t2	local_a=A;	
t3		A=10;
t4		B=10;
t5		TX_END(1);
t6	local_b=B;	
t7	while(local_a!=local_b)	
	TX_END(0);	

図 1 Opacity が保証されないことによる無限ループ

この問題について，**図 1** のように，同じ共有変数に対してアクセスする Tx が異なるスレッドによって実行される場合を例に説明する．なお，図 1 中の TX_BEGIN() および TX_END() はそれぞれ Tx の開始および終了を表している．また，プログラム内の Tx はそれぞれ固有の ID を持ち，これらの関数の引数は当該 Tx の ID を表している．また，共有変数の初期状態を A=B=0 とする．

まず，2 つのスレッドが Tx の実行を開始し（t1），*Thread*1 が共有変数 A の値 0 をローカル変数 local_a へ代入する（t2）．次に，*Thread*2 が共有変数 A および B に値 10 を代入し（t3,t4），Tx をコミットしたとする（t5）．次に，*Thread*1 が共有変数 B を読み出すが，直前に *Thread*2 が共有変数 B を更新しているため，値 10 が読み出され，local_b に代入する（t6）．そして，*Thread*1 がローカル変数 local_a の値とローカル変数 local_b の値が一致しているかを確認し，無限ループに陥ってしまう（t7）．

ここで，図 1 の Tx をそれぞれ逐次実行した場合，*Thread*1 が読み出す共有変数の値は A=B=0 もしくは A=B=10 となる．しかし，この例では *Thread*1 が A=0 と B=10 という逐次実行した場合では起こり得ない値のペアを読みだしてしまっており，Opacity の要件を満たしていない．

2.3 STM における一貫性検証

STM では，ソフトウェアにより実装されることによる柔軟性から多様なアルゴリズムが実現できるため，近年さまざまな一貫性検証手法が開発されている [9]．本稿では，その中でもタイムスタンプベース検証とバージョンベース検証に注目し，本節においてそれぞれの検証手法について説明する．

2.3.1 タイムスタンプベース検証

STM の実装の一つでは，システム全体で論理時刻を保持する**グローバルクロック**を用意し，このグローバルクロックが示す時刻を用いて共有変数の一貫性検証を行う．この検証方法を**タイムスタンプベース検証**と呼ぶ．なお，このグローバルクロックは Tx の開始時に読み出され，Tx のコミット時にインクリメントされる．また，各共有変数はメタデータを持ち，このメタデータには最後に当該共有変数

```
1 TX_BEGIN(0);
2 int tmp1 = TX_SHARED_READ(x);
3 int tmp2 = TX_SHARED_READ(y);
4 TX_SHARED_WRITE(x, tmp1 + tmp2);
5 TX_END(0);
```

図 2 サンプルプログラム

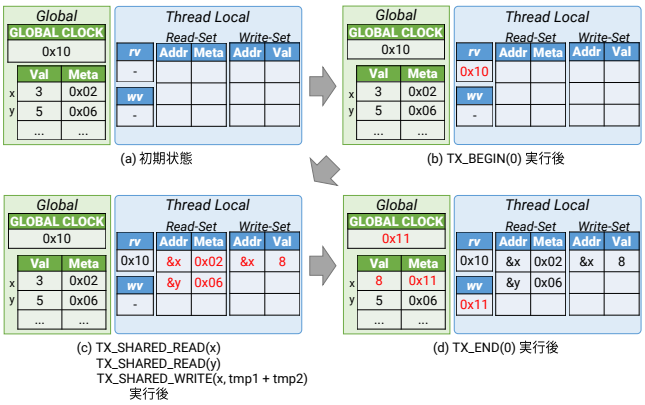

図 3 タイムスタンプベース検証を用いた場合のメモリの状態変化

が更新された際のグローバルクロックの値が記録される．タイムスタンプベース検証では，このメタデータが Tx 実行開始時のグローバルクロックより大きい，つまり当該 Tx 開始時以降に他の Tx によって更新が行われたことを検知した際，これを競合として検出する．

ここで，タイムスタンプベース検証における Tx の動作について，**図 2** のサンプルプログラムと**図 3** とを用いて説明する．なお，サンプルプログラム中の TX_SHARED_READ()，TX_SHARED_WRITE() 関数はそれぞれ Tx 内で共有変数に対して読み出し，書き込みを行う関数である．また，各 Tx には ***Read-Set*** と ***Write-Set*** と呼ばれるバッファが保持され，*Read-Set* には読み出しがあった共有変数のアドレスとそのメタデータとのペアが，*Write-Set* には書き込みがあった共有変数のアドレスと書き込む値とのペアが記録される．

はじめに，1 行目の Tx 実行開始時にはグローバルクロックの値を読み出し，スレッドローカルな変数に記録する（図 3(b)）．このスレッドローカルな変数は **Read Version**（***rv***）と呼ばれる．次に，2 行目の TX_SHARED_READ 関数により共有変数 x の値が読み出されるが，このときタイムスタンプベース検証では共有変数 x のメタデータが *rv* 以下であること，つまり Tx の開始時以降に他の Tx が当該共有変数に対し書き込みを行っていないことを確認する．その後，共有変数 x のアドレスとそのメタデータとのペアを *Read-Set* に記録する．同様に，3 行目の TX_SHARED_READ 関数でも，共有変数 y の値を読み出すと同時に，共有変数 y のメタデータが *rv* 以下であることを確認し，*Read-Set* に記録する．なお，メタデータが *rv* より大きかった場合は Tx をアボートし，再実行する．こ

```
1 function IncrementAndFetch(address location){
2   (*location)++;
3   return *location;
4 }
```

図 4 IncrementAndFetch

のように，タイムスタンプベース検証では *Read-Set* の各エントリが Tx 開始時のスナップショットと等価であることを保証する．次に，4 行目の TX_SHARED_WRITE 関数により共有変数 x に値を書き込むが，このとき STM では直接グローバルな領域に書き込まず，スレッドローカルな領域に保持される *Write-Set* に，共有変数 x に書き込む値とそのアドレスとのペアを記録する（図 3(c)）．最後に，5 行目の Tx 終了時では，まず *Write-Set* の各エントリに対応するロックを取得し，*Read-Set* 内の各エントリのアドレスに対応するグローバルな領域のメタデータが *rv* 以下であることを確認する．なお，このとき *Write-Set* の各エントリに対応するロックを取得出来なかった場合や，*Read-Set* のいずれかのエントリのアドレスに対応するグローバルな領域のメタデータが *rv* より大きかった場合，Tx をアボートし，再実行する．そして，*Write-Set* の各エントリをグローバル領域の対応するアドレスへ上書きする．その後，この時点におけるグローバルクロックの値をインクリメントアンドフェッチした値を，Tx 内で更新した共有変数に対応するメタデータ全てに上書きする．この値は **Write Version**（***wv***）と呼ばれる．ここで，インクリメントアンドフェッチとは，**図 4** に示す疑似コードのような処理をアトミックに行う命令を指す．その後，取得したロックを解放することで Tx のコミットを完了する（図 3(d)）．

タイムスタンプベース検証では以上のように動作することで Tx の性質と Opacity を保証するが，グローバルクロックは単一の共有カウンタを用いて実装されるため，数多くの Tx が同時に実行されるようなワークロードでは，このグローバルクロックに対するアクセスがボトルネックとなりうるという問題がある．

2.3.2 バージョンベース検証

グローバルクロックを用いない一貫性検証手法として**バージョンベース検証**がある．バージョンベース検証では，各共有変数が個別にカウンタをメタデータとして持ち，これは当該共有変数が更新された際にインクリメントされる．バージョンベース検証では，Tx 終了時のメタデータが Tx 内で当該共有変数を最初に読み出した際のメタデータと異なっていた場合，これを競合として検出する．

ここで，バージョンベース検証における Tx の動作について，図 2 のサンプルプログラムと**図 5** とを用いて説明する．なお，バージョンベース検証においても各 Tx には *Read-Set* と *Write-Set* とが保持される．

はじめに，1 行目の Tx 実行開始時には，特に何も行われ

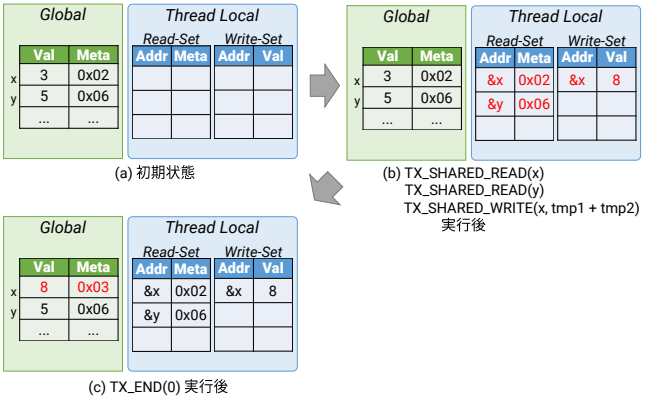

図 5 バージョンベース検証を用いた場合のメモリの状態変化

ない．次に，2 行目の TX_SHARED_READ 関数では，共有変数 x の値が読み出され，共有変数 x の値とそのアドレス，バージョン情報との組を *Read-Set* に記録する．同様に，3 行目の TX_SHARED_READ 関数でも，共有変数 y の値が読み出され，その値とアドレス，バージョン情報との組を *Read-Set* に記録する．ここで，2 行目と 3 行目の間で他の Tx が共有変数 x に対して書き込みを行っていた場合を考える．この場合バージョンベース検証だけではこの書き込みを検知できず，Opacity を保証できない．そこで，バージョンベース検証では共有変数の読み出し時に，当該 Tx 内でそれまでに読み出した共有変数すべてに対し，最初に読み出した際に *Read-Set* に記録したバージョン情報と現在の当該共有変数に対応するメタデータとを比較し，差異がないことを確認する．これを**増分検証（Incremental Validation）**と呼ぶ．増分検証により，*Read-Set* 内のエントリが増分検証実行時のスナップショットと等価であることが保証できる．次に，4 行目の TX_SHARED_WRITE 関数では，タイムスタンプベース検証と同様に *Write-Set* に共有変数 x に書き込む値とそのアドレスとのペアを記録する（図 5(b)）．最後に，5 行目の Tx 終了時では，タイムスタンプベース検証と同様に *Write-Set* の各エントリに対応するロックを取得し，*Read-Set* 内の各エントリのメタデータと Tx 内で最初に読み出した際に記録したバージョン情報とに差異がないか確認した後，*Write-Set* の各エントリをグローバル領域の対応するアドレスへ上書きする．その後，*Write-Set* の各エントリのメタデータをインクリメントし，取得したロックを解放することで Tx のコミットを完了する（図 5(c)）．

バージョンベース検証では，このように動作することで Tx の性質と Opacity を保証する．この方法では，タイムスタンプベース検証におけるグローバルクロックのような単一の共有カウンタに対するアクセスが存在しないため，スケーラビリティに優れる．しかし，Opacity を保証するために増分検証が必要となり，この増分検証には *Read-Set* のエントリ数 n に対して $O(n^2)$ の計算量を要するため，*Read-Set* のエントリ数が多くなればなるほどオーバヘッドが膨大となるという問題がある．

```
1 int global_array[N]
2
3 TX_BEGIN();
4 for(int i = 0; i < READSETSIZE; ++i){
5   TX_SHARED_READ(global_array[i])
6 }
7 ...
8 TX_END();
```

図 6 調査用プログラム

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 5.4.0
Compile options	-O3

3. 調査

2.3 節で述べたように，タイムスタンプベース検証では単一の共有カウンタに対するアクセスがボトルネックとなる可能性があり，バージョンベース検証では増分検証が大きなオーバヘッドを発生させる可能性がある．本章では，それぞれの検証手法において Tx 内における共有変数に対する読み出し回数，つまり *Read-Set* のエントリ数を変更し，これが性能に与える影響を調査した結果を示す．

3.1 調査環境

調査環境を**表 1** に示す．また，調査に用いたプログラムの疑似コードを**図 6** に示す．なお，STM には TL2[3] を採用した．TL2 ではタイムスタンプベース検証が用いられているため，バージョンベース検証も TL2 上に実装し，これらを比較した．

3.2 調査結果

調査用プログラムを 32 スレッドで実行した調査結果を**図 7** のグラフに示す．グラフの縦軸は毎秒あたりの平均 Tx 実行回数（Tx/s）を，横軸は Tx を実行した際の *Read-Set* のエントリ数を表す．また，グラフの黒線はタイムスタンプベース検証を，青線はバージョンベース検証を用いた際の Tx/s をそれぞれ表している．

まず，*Read-Set* のエントリ数が小さい時の結果に着目すると，バージョンベース検証がタイムスタンプベース検証の性能を大きく上回っていることが確認できる．次に，*Read-Set* のエントリ数を大きくした際の結果に着目する

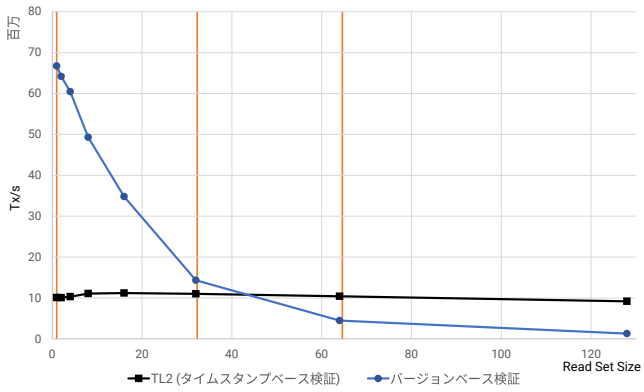

図 7　32 スレッドにおける実行結果

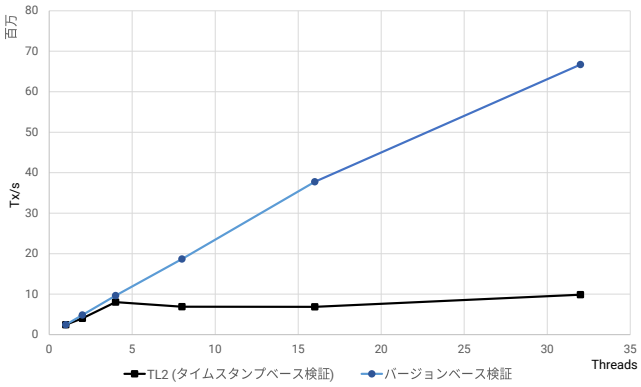

図 8　*Read-Set Size* = 1

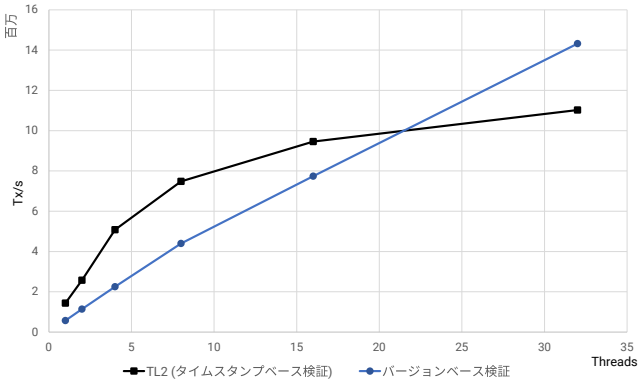

図 9　*Read-Set Size* = 32

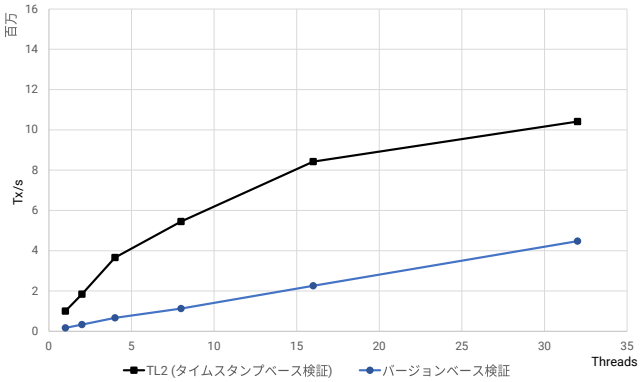

図 10　*Read-Set Size* = 64

と，タイムスタンプベース検証では性能にほとんど変化がないのに対し，バージョンベース検証では *Read-Set* のエントリ数の増大に伴い性能が低下してゆき，*Read-Set* のエントリ数が 40 前後になるあたりを境にタイムスタンプベース検証の性能を下回ることが確認できる．

また，*Read-Set* のエントリ数を 1，32，64 に固定し 1，2，4，8，16，32 スレッドで実行した結果を，それぞれ**図 8**，**図 9**，**図 10** のグラフに示す．グラフの縦軸は毎秒あたりの平均 Tx 実行回数（Tx/s）を，横軸はプログラムを実行した際のスレッド数を表す．まず，*Read-Set* のエントリ数が 1 の場合の結果（図 8）に着目すると，バージョンベース検証ではスレッド数の増加に比例して性能が向上していくが，タイムスタンプベース検証はあるスレッド数から性能が横ばいになることが確認できる．その結果，*Read-Set* のエントリ数が 1 の場合では，4 スレッド以上で実行した場合においてバージョンベース検証がタイムスタンプベース検証の性能を大きく上回っている．一方，*Read-Set* のエントリ数を 32，64 に増加させた場合の結果（図 9，図 10）では，バージョンベース検証では変わらずスレッド数の増加に比例して性能が向上していくが，その傾きは *Read-Set* のエントリ数が多いほど小さくなっている．一方，タイムスタンプベース検証では，*Read-Set* のエントリ数による性能の変化はわずかである．その結果，*Read-Set* のエントリ数が 32 の場合には，図 9 に示すようにあるスレッド数を境にタイムスタンプベース検証とバージョンベース検証の性能が逆転しているが，*Read-Set* エントリ数が 64 と十分大きい場合の図 10 では，32 スレッド未満の範囲ではタイムスタンプベース検証の方が常に良い性能を示している．このように，いずれの検証手法が適しているかは，*Read-Set* のエントリ数とスレッド数とに依存して決まると考えられる．

4.　一貫性検証の動的切り替え

3 章で示した調査結果により，ワークロードごとにそれぞれ有利な一貫性検証手法は異なることが確認できた．そこで本稿では，Tx ごとに適切な検証手法を判定しこれを動的に切り替える手法を提案する．本章では検証手法の切り替え基準とその動作，複数の検証手法が混在することにより発生する問題とその解決方法について述べる．

4.1　検証手法の切り替え基準とタイミング

3 章の調査結果より，*Read-Set* のエントリ数と実行するスレッド数とによって適切な一貫性検証手法は異なることが分かった．そこで本稿では，プログラム実行開始時はすべての Tx の一貫性検証をバージョンベース検証で行い，*Read-Set* のエントリ数が閾値を超えた際に，当該 Tx をタイムスタンプベース検証に切り替える手法を提案する．なお，一貫性検証を切り替える閾値として，経験則により得

表 2　メタデータ

Global Metadata
Global Clock
Validate Mode [NUM_TX]
Valiable Metadata
Version
Validate Mode

た一定の値に実行するスレッド数を乗算した値を用いる．

以上の動作を行うため，バージョンベース検証で動作する Tx を実行する各スレッドは共有変数に対する読み出し時に，自身の *Read-Set* のエントリ数を検査する．ここで，*Read-Set* のエントリ数が閾値を超えていた場合，当該 Tx を実行するスレッドは自身の検証手法をタイムスタンプベース検証に切り替え，アボートする．

4.2　メタデータの管理

提案手法では，Tx ごとに一貫性検証手法を切り替えるため，各 Tx がどちらの検証手法で動作しているかを示すグローバルメタデータを用意する（**表 2**）．各スレッドは Tx の実行を開始する際，当該 Tx に対応するグローバルメタデータを調べ，それに基づいた検証手法で動作する．また，共有変数のメタデータとして記録すべき情報は，タイムスタンプベース検証では，最後に当該共有変数が更新された際のグローバルクロックの値であり，バージョンベース検証では，当該共有変数が何度更新されたかを表す値である．これらを区別するため，各共有変数がどちらの検証手法で管理されているかを示す情報を，メタデータに付加する．以降では，ある変数 X がどちらの検証手法で管理されているかを示すメタデータを $\mathbf{Mode}_X$，また，X のバージョン情報を記録しているメタデータを $\mathbf{Ver}_X$ と定義する．Mode_X はタイムスタンプベース検証を表す値である t，もしくはバージョンベース検証を表す値である v のいずれかの値をとるものとする．

4.3　共有変数への読み出し，書き込み

本節では，提案手法による共有変数に対する読み出しと書き込み（Tx のコミット）の動作について説明する．

4.3.1　共有変数に対する読み出し

提案手法における，共有変数に対する読み出しを行う関数の疑似コードを**図 11** に示す．ここで，タイムスタンプベース検証により動作する Tx を $\mathbf{Tx}_T$，バージョンベース検証により動作する Tx を $\mathbf{Tx}_V$ と定義する．なお，Tx 開始時におけるグローバルクロックの読み出しおよび Tx 終了時におけるグローバルクロックのインクリメントは Tx_T のみが行う．

まず，Tx_T が共有変数を読み出す場合を考える．共有変数に対する読み出しでは，読み出し対象である共有変数に対応するメタデータを読み出してからその共有変数の値を

```
1  function TX_SHARED_READ(address addr) {
2    ...
3    // Read Metadata
4    meta = GetMetadata(addr);
5    // Read from Global Memory
6    value = ReadfromGlobal(addr);
7    if (meta != GetMetadata(addr)) {
8      TX_ABORT();
9    }
10   // Validate Consistency
11   if (GetTxMode(TxID) == t) {
12     if (meta.mode == t) {
13       if (meta.ver > rv) {
14         TX_ABORT();
15       }
16     } else {
17       if (!IncrementalValidation()) {
18         TX_ABORT();
19       }
20     }
21   } else {
22     if (!IncrementalValidation()) {
23       TX_ABORT();
24     }
25   }
26   // Add to Read Set
27   AddReadSet(addr, mata);
28   return value;
29 }
```

図 11　提案手法における共有変数に対する読み出し

読み出す（line 3-6）．値を読み出した後再度メタデータを読み出し，一度目に読み出したものとの差異を確認することで，メタデータを読み出してから値を読み出すまでに他の Tx により当該共有変数に対する書き込みが発生していないかを検査する（line 7）．その後，読み出し対象である共有変数がどちらの検証手法で管理されているかを判断する（line 11）．読み出し対象である共有変数 X の検証手法を表すメタデータ Mode_X の値が t であるなら，通常通り Ver_X が自身の *rv* 以下であることを確認する（line 13）．しかし，Mode_X の値が v である場合，Ver_X に記録されている情報がグローバルクロックの値でないため，Tx 開始時以降の他 Tx による当該共有変数に対する書き込みの有無を検出する手段がなく，Opacity の要件を満たせない可能性がある．そのため，Tx_T がバージョンベース検証で管理されている共有変数を読み出す場合，増分検証を行う必要がある（line 18）．その後，*Read-Set* に共有変数のアドレスとメタデータとのペアを記録し，関数を終了する（line 26-28）．

次に，Tx_V が Tx 内で共有変数を読み出す場合を考える．Tx_T の場合と同様に，値の読み出しを行う前後でメタデータを二度読み出して他の Tx による当該共有変数に対する書き込みが発生していないかを検査する（line 3-9）．その後，共有変数の一貫性を検証するが，Tx_V ではタイムスタンプベース検証で管理されている共有変数を読み出す場合と，バージョンベース検証で管理されている共有変数を読み出す場合のいずれにおいても，増分検証を行えば問題は

```
1  function TX_END(TxID) {
2    // Lock Write Set
3    foreach (entry e in writeSet) {
4      if (!Lock(e)) {
5        // unable lock
6        TX_ABORT();
7      }
8    }
9    // Validate Read Set
10   foreach (entry e in readSet) {
11     meta = GetMetadata(e.addr);
12     if (GetTxMode(TxID) == t) {
13       if (meta.mode == t) {
14         if (meta.ver > rv) {
15           TX_ABORT();
16         }
17       } else {
18         if (meta.ver != e.ver) {
19           TX_ABORT();
20         }
21       }
22     } else {
23       if (meta.ver != e.ver) {
24         TX_ABORT();
25       }
26     }
27   }
28   if (GetTxMode(TxID) == t) {
29     // Acquire unique WV
30     wv = IncrementAndFetch(GLOBALCLOCK);
31   }
32   // Write back updates
33   foreach (entry e in writeSet) {
34     meta = GetMetadata(e.addr);
35     if (GetTxMode(TxID) == t) {
36       meta.ver = wv;
37       meta.mode = t;
38     } else {
39       meta.ver++;
40       meta.mode = v;
41     }
42     WriteBackGlobal(e.addr, e.value);
43     Unlock(e);
44   }
45 }
```

図 12 提案手法における共有変数に対する書き込み

発生しない（line 17-19）．これは，Tx_V がタイムスタンプベース検証で管理されている共有変数 X を読み出したとしても，Ver_X に記録されているグローバルクロックの値をカウンタとして見なすことで，増分検証時および Tx 終了時に，その時点におけるカウンタと最初に読み出した際のカウンタとの差異を判定することで競合を検出できるためである．

4.3.2 共有変数に対する書き込み

次に，提案手法における共有変数に対する書き込み動作（コミット動作）について説明する．この疑似コードを**図 12** に示す．

Tx_T がコミットされる場合はまず *Write-Set* の各エントリに対応するロックを獲得する（line 2-8）．すべてのロックを獲得した後，*Read-Set* の各エントリの一貫性を検証する（line 9-27）．*Read-Set* に登録されている共有変数 X の検証手法を表すメタデータ Mode_X の値が t であるなら，

*Thread*1	*Thread*2
TX_BEGIN(0);	TX_BEGIN(1);
	X=10;
	TX_END(1);
local=X; // false conflict	
TX_END(0);	

図 13 タイムスタンプベース検証における偽の競合

Ver_X が自身の *rv* 以下であることを確認する（line 14）．Mode_X が v であるなら，当該エントリを読み出した際に *Read-Set* に記録した Ver_X と現在の Ver_X とを比較し，差異がないかを確認する（line 18）．その後，*wv* を発行し（line 30），*Write-Set* の各エントリのバージョン情報，および，検証手法を示すメタデータとして，それぞれ *wv*，および，タイムスタンプベース検証で管理されていることを示す値である t を記録し（line 36-37），当該エントリに記録されている値を対応するグローバルな領域へ上書きしたうえで（line 42），ロックを解放する（line 43）ことで Tx のコミットを完了する．

次に，Tx_V がコミットされる場合を考える．Tx_T と同様に，Tx のコミット時はまず *Write-Set* の各エントリに対応するロックを獲得する（line 2-8）．その後，*Read-Set* の各エントリ X に対し，当該エントリを読み出した際に *Read-Set* に記録した Ver_X と現在の Ver_X とを比較し差異がないかを確認することで *Read-Set* の一貫性を検証する（line 18）．なお，Tx_V ではグローバルクロックを用いないため，*wv* の発行に伴うグローバルクロックに対する書き込みを省略できる（line 28-31）．その後，*Write-Set* の各エントリに対し，対応するメタデータのバージョン情報のインクリメント（line 39）と検証手法を示す情報にバージョンベース検証で管理されていることを示す v の記録（line 40）とを行い，当該エントリに記録されている値をグローバルな領域へ上書きし（line 42），ロックを解放する（line 43）ことで Tx のコミットを完了する．

なお，異なる検証手法で動作する Tx によって同一共有変数 X が更新される場合，Mode_X の変化が単調増加とならないことで，競合が検出できなくなる可能性がある．そこで，Tx_T がコミット時に発行する *wv* の値は，当該 Tx が読み出したバージョン情報の最大値+1 とする．

また，タイムスタンプベース検証では本来競合として扱う必要がない**偽の競合**が発生し得る（**図 13**）．提案手法ではこの偽の競合を回避するため，文献 [10] を参考に，読み出した共有変数 X の Ver_X が *rv* より大きかった場合に，増分検証を行い，*Read-Set* の一貫性が保たれていることが確認できた場合にのみ，*rv* を更新するという処理を行う（**図 14**）．

5. 評価

提案手法の有効性を検証するため，前章で述べた手法を

```
1  function TX_SHARED_READ() {
2    ...
3    if (meta.ver > rv) {
4      tmp_rv = GLOBALCLOCK;
5      if (!IncrementalValidation()) {
6        TX_ABORT();
7      }
8      // update rv
9      rv = tmp_rv;
10   }
11   ...
12 }
```

図 14 *rv* の更新

表 3 ベンチマークパラメタ

Genome	-g16384 -s64 -n16777216
Intruder	-a10 -l128 -n262144
Kmeans	-m40 -n40 -t0.00001 -i random-n2048-d16-c16.txt
Labyrinth	-i random-x512-y512-z7-n512.txt
Ssca2	-s20 -i1.0 -u1.0 -l3 -p3
Yada	-a15 -i ttimeu1000000.2

TL2 に実装し，評価を行った．本章ではその評価結果を示し考察する．

5.1 評価環境

実行環境は 3 章の調査で用いたものと同様である．評価対象のプログラムとしては，TM の研究に広く用いられている STAMP[11] から Genome，Intruder，Kmeans，Labyrinth，Ssca2，Yada の 6 種のベンチマークを用いた．実行に用いた入力パラメタを**表 3** に示す．なお，各ベンチマークプログラムはそれぞれ 32 スレッドで実行した．

5.2 評価結果

図 15 は，各ベンチマークのプログラムの評価結果を表している．図中の 3 本のバーは左から順に，

(T) タイムスタンプベース検証で動作する既存の TL2（ベースライン）

(V) TL2 をバージョンベース検証で動作するよう改変した参考モデル

(P) 提案モデル

を表しており，ベースライン (T) の実行時間を 1 として正規化している．評価結果より，提案モデル (P) はベースライン (T) に対して全てのベンチマークで性能が向上し，最大 27.0%，平均 15.3%の実行時間削減を達成した．また，バージョンベース検証を用いる (V) で，ベースライン (T) より性能が悪化している，Genome，Kmeans，Yada においても，提案モデル (P) による性能向上が確認できることから，Tx ごとに適切な検証手法を適用することの有効性が確認できた．しかし，Labyrinth では参考モデル (T) ほどの性能を達成できていない．これは，Labyrinth に含まれるある Tx では，その末尾でまとめて共有変数の読み出しを行うため，適切な検証手法を判断するまでに時間がかかったことで，切り替えに伴うアボートが性能低下を引き起こしたためであると考えられる．

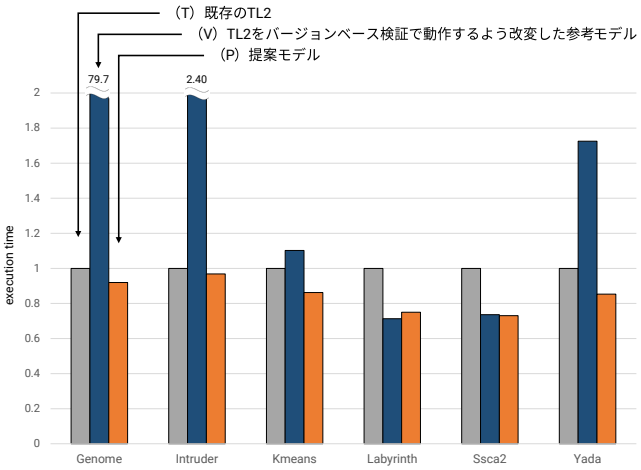

図 15 各プログラムにおける実行時間比

6. おわりに

本稿では，STM で用いられる一貫性検証手法である，タイムスタンプベース検証とバージョンベース検証の特徴について調査した．タイムスタンプベース検証ではグローバルクロックに対するアクセスがボトルネックとなる可能性があり，バージョンベース検証では Tx 内における共有変数に対する読み出し回数が多くなればなるほどオーバヘッドが膨大となるという問題がある．そこで，実行時のスレッド数と Tx 内で読み出す共有変数の個数とを基準に，Tx ごとに一貫性検証手法を適切なものへ切り替える手法を提案し，STAMP ベンチマークプログラムを用いて評価を行った．その結果，既存の STM と比較して 32 スレッドで最大 27.0%，平均 15.3%の実行時間を削減できることを確認した．今後の課題として，より多様なベンチマークプログラムを用いた評価，および，よりスケーラビリティが重要となる環境として GPGPU 向けの拡張を行うことなどが挙げられる．

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] Shavit, N. et al.: Software Transactional Memory, *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pp. 204–213 (1995).

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

[4] 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).

[5] Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C. and Hertzberg, B.: McRT-STM: a High-Performance Software Transactional Memory System for a Multi-Core Runtime, *Proc. 11th ACM SIGPLAN Symp. on Princibles and Practice of Parallel Programming (PPoPP'06)*, pp. 187–197 (2006).

[6] Harris, T., Plesko, M., Shinnar, A. and Tarditi, D.: Optimizing Memory Transactions, *Proc. 27th ACM SIGPLAN Conf. on Programming Language Design and Implemantation (PLDI'06)*, pp. 14–25 (2006).

[7] Guerraoui, R. and Kapalka, M.: On the Correctness of Transactional Memory, *Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'08)*, pp. 175–184 (2008).

[8] Dziuma, D., Fatourou, P. and Kanellou, E.: Consistency for Transactional Memory Computing, *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, Springer, pp. 3–31 (2015).

[9] Spear, M. F., Marathe, V. J., Scherer, W. N. and Scott, M. L.: Conflict Detection and Validation Strategies for Software Transactional Memory, *Int'l Symp. on Distributed Computing*, Springer, pp. 179–193 (2006).

[10] Riegel, T., Fetzer, C. and Felber, P.: Time-based transactional memory with scalable time bases, *Proc. 19th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, ACM, pp. 221–228 (2007).

[11] 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).