

軽量な一貫性保証と高並列性の両立に向けた トランザクショナルメモリの検証アルゴリズム分析

白坂 亘¹ 佐藤 宏樹¹ 藤井 創悟¹ 伊原 楓¹ 小泉 透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) では一貫性保証にかかるコストが大きく、ロックに基づくプログラミング (LBP) と比べて性能が劣る場合がある。低コストな一貫性保証を実現する実装として InvalSTM が提案されているが、トランザクションの並列実行を過度に制限しており、並列性を損ねている。本稿では、細粒度ロックを基盤とする一般的な TM をベースに、InvalSTM の無効化方式による低コストな一貫性保証を統合することで、軽量な一貫性保証と高い並列性を両立する提案手法を通じ、無効化方式を細粒度ロック環境に導入した際のボトルネックを、検証コストのスケラビリティ、Bloom filter 操作のオーバーヘッド、ロック保持期間とアボート率との関係などの側面から定量的に解明し、将来の高性能 STM 設計に向けた実装指針を提示する。

1. はじめに

共有メモリ型の並列プログラミングでは、共有変数に対するアクセスを調停する必要がある。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、ロックに基づくプログラミング (Lock-Based Programming: LBP) は、ロック操作のオーバー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プログラムの正当性を損なう問題が生じる恐れがある。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) [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る。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処理を、トランザクション (以降、Tx と省略する場合がある) 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 Tx の実行が投機的であるため、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。これを競合検出という。TM のハードウェア実装であるハードウェアアトランザクショナルメモリ (Hardware Transactional Memory: HTM) [2] では、競合検出の機構をハードウェアで実現することで、Tx の実行によるオーバーヘッドを抑制している。一方、TM のソフトウェア実装であるソフトウェアアトランザクショナルメモリ (Software Transactional Memory: STM) [3] では、競合検出の機構をソフトウェアで実現することで、ハードウェアの制約を受けずに Tx 実行が可能である。

さらに、TM では、競合検出に加えて、プログラマの意図しない挙動の発生を防ぐため、一貫性保証を行う必要がある。しかし、TinySTM [4] のような一般的な TM では、一貫性保証にかかるコストが大きい。そのため、実行するワークロードによっては、TM の性能が LBP と比べて劣ってしまうことがある。一方で、InvalSTM [5] という TM 実装では、一般的な TM とは異なる一貫性保証手法を採用することで他の実装と比べて、Tx 実行中のコストが小さい一貫性保証を実現している。しかし、この実装では Tx の並列実行を必要以上に制限しており、並列性を損ねてしまっている。

そこで本稿では、TinySTM に代表される一般的な TM をベースとして、その一貫性保証手続きに InvalSTM の無効化 (Invalidation) 方式を採用することで、軽量な一貫性

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

保証と高並列性を両立する TM の設計を提案する。そして、その評価を通じて、無効化方式を細粒度ロック環境に導入した際のボトルネックを、検証コストのスケラビリティ、Bloom filter 操作のオーバーヘッド、ロック保持期間とアボート率との関係などの側面から定量的に解明し、将来の高性能 STM 設計に向けた実装指針を提示する。

2. トランザクショナルメモリ

本章では、本研究で扱う TM の概要について述べる。

2.1 競合検出

TM では、Tx 同士を投機的に並列実行した際に共有変数の整合性を保つために、並列実行の結果が逐次実行した場合の結果と等価であるという性質 (**Serializability: 直列化可能性**) を保証しなければならない。そのため、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るか、もしくは全く実行されないかのいずれかであり、各 Tx 内における処理結果は、Tx の終了と同時に観測される。

Consistency (一貫性) : Tx の実行前後で、共有変数の一貫性が保たれる。

Isolation (排他性) : Tx の中間状態が隠蔽され、他の Tx から観測されない。

TM では、投機的に Tx を実行するために、共有変数へのアクセスを監視するための機能を持つ。そして、Tx 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。この操作を**競合検出 (Conflict Detection)** と呼び、以下の 3 パターンのアクセスが競合として検出される。

Read after Write (RaW) : Tx を実行中のスレッドがある共有変数へ Write アクセスしてから当該 Tx の実行を完了するまでの間に、異なるスレッドが同一共有変数へ Read アクセスするパターン。この Read アクセスが許可されると、Tx の実行が完了する前に値を読み出されてしまうことで、本来読み出されるべき値とは異なる値が読み出されてしまう可能性があり、Serializability が損なわれる。

Write after Read (WaR) : Tx を実行中のスレッドがある共有変数へ Read アクセスしてから当該 Tx の実行を完了するまでの間に、異なるスレッドが同一共有変数へ Write アクセスするパターン。この Write アクセスが許可されると、Read アクセスした Tx の実行を完了する前に共有変数の値が上書きされ、Read アクセスした Tx が再度、共有変数の値を読み出した場合、一貫性が損なわれる。

Write after Write (WaW) : Tx を実行中のスレッドがある共有変数へ Write アクセスしてから当該 Tx

の実行を完了するまでの間に、異なるスレッドが同一共有変数へ Write アクセスするパターン。この Write アクセスが許可されると、Write アクセス済の Tx が途中まで実行された状態において、その値の変更という他の処理が実行されたことになり、一貫性が損なわれる。

これらのアクセスパターンを競合として検出するために、TM では、Tx 内でアクセスされた変数のアドレスを記憶する。そのうち、Read アクセスされた変数のアドレスを記憶する領域を **Read-set**、Write アクセスされた変数のアドレスを記憶する領域を **Write-set** という。

競合が検出された場合、一貫性を保証するため、競合の原因となっただけの Tx の実行を中止する。これを**アボート (Abort)** という。また、Tx をアボートしたスレッドは、Tx 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。この操作を**ロールバック (Rollback)** という。なお、Tx が終了するまで競合が検出されず、その処理が最後まで完了した場合、Tx 内で行ったデータの更新が確定される。この操作を**コミット (Commit)** という。

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) : Tx 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) : Tx のコミットを試みる時点で、その Tx 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 Tx 実行が無駄になるだけであるが、LazyCD では Tx 全体の実行が無駄になってしまう。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 よりオーバーヘッドが小さく済む。

なお、Tx をアボートした際に Tx 開始前の状態を復元するためには、Tx 内で更新したデータと更新する前のデータの両方を保持しておく必要がある。そこで、TM では Tx 内で更新したデータ、もしくは更新前のデータをそのアドレスとともに別領域に退避する。このようなデータの管理を**バージョン管理 (Version Management)** という。バージョン管理はどちらの値をメモリに残し、どちらの値を別領域に退避させるかによって以下の 2 つの方式に大別される。

Eager Version Management (EagerVM) : 更新前の値を別領域にバックアップし、更新後の値をメモリに上書きする。コミットは別領域にバックアップした値を破棄するだけで実現できるためオーバーヘッドが小

さいが、アポート時にはバックアップした値をメモリに書き戻す必要があるため、オーバーヘッドが大きい。

Lazy Version Management (LazyVM) : 更新前の値をメモリに残し、更新後の値を別領域にバッファする。アポートは別領域にバッファした値を破棄するだけで実現できるためオーバーヘッドが小さいが、コミットはバッファした値をメモリに上書きする必要があるため、オーバーヘッドが大きい。

このように TM は、競合検出機能とバージョン管理機能を備えることで、Tx を投機的に実行することができ、競合が発生しない限り Tx は並列に実行される。このため、競合がなくてもクリティカルセクションが排他的に実行される LBP に対して高い性能が期待できる。

2.2 Opacity

2.1 節で述べたように、TM は Serializability を満たす必要があるが、Tx の一貫性の基準として Serializability だけでは必ずしも十分ではない。Serializability は、あくまで Tx の実行結果の一貫性について言及したものにすぎず、Tx 内の各ステップにおける実行結果の一貫性には言及していないためである。このため、TM における Tx 内の各ステップにおける実行結果の一貫性について定めた基準として **Opacity** [6] が提唱されている。

Opacity とは、Tx は常にシステムの一貫したメモリビューを観測しなければならない、という性質である。これは、Tx 内で読み出したすべての共有変数の値は、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あとの値とを混在させて読み出す可能性があり、これは予期せぬ無限ループやセグメンテーション違反といった、プログラマの意図しない動作を引き起こす原因となる。

この問題について、図 1 のコード例を用いて説明する。なお、図 1 におけるマクロ 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、これらのマクロの引数は各 Tx 固有の ID を示している。この例は、同じ共有変数に対してアクセスする Tx が異なるスレッドで実行される状況を示している。共有変数 a, b について、初期状態が $a = b = 0$ であるとする (t0)。まず、2つのスレッドが Tx の実行を開始し (t1, t2)、Thr.0 が共有変数 a の値を読み出してローカル変数 tmp_a に 0 を代入する (t2)。次に、Thr.1 が共有変数 a および b に 1 を書き込み (t3, t4)、この変更をコミットする (t5)。その後、Thr.0 が共有変数 b の値を読み出す。このとき、すでに共有変数 b の値は 1 に更新されているため、Thr.1 は読み出した値 1 をローカル変数 tmp_b に代入する (t6)。すると、tmp_a の値と tmp_b の値が一致しないため、Thr.0 は無限ループに陥る (t7)。

Time	Thr.0	Thr.1
t0	$a = 0; b = 0;$	
t1	TX_BEGIN(0);	
t2	tmp_a = a;	TX_BEGIN(1);
t3		$a = 1;$
t4		$b = 1;$
t5		TX_END(1);
t6	tmp_b = b;	
t7	while(tmp_a != tmp_b){}	
	TX_END(0);	

図 1: Opacity が保証されない場合に無限ループに陥る例

ここで、図 1 の Tx をそれぞれ逐次実行する場合を考えると、Thr.0 が読み出す共有変数 a, b の値の組は (0, 0) もしくは (1, 1) 以外にはなりえず、無限ループに陥ることはない。しかし、この例では Thr.0 が (0, 1) という、逐次実行した場合ではありえない値の組を読み出してしまっており、Opacity を保証できていない。Opacity を保証していない場合、Tx は分離して実行されるはずというプログラマの期待に反する動作をする可能性があるため、TM において Opacity を保証することは重要である。

Opacity を保証するには、各共有変数に対して Read アクセスを行ったスレッドや Write アクセスを行ったスレッドを管理し、不正なアクセスを行おうとする Tx をアポートさせる必要がある。

3. 既存の STM 実装とその問題点

本章では、既存の STM 実装である、TinySTM と Inval-STM について説明し、その問題点を述べる。

3.1 TinySTM

3.1.1 競合検出

TinySTM での競合検出は、共有変数への Write アクセス時に、当該変数に用意されている**共有変数ごとのロック**を獲得し、変数へのアクセス時にそのロックを確認する、EagerCD 方式である。

この手法では、共有変数へのアクセス時、当該変数のロックが獲得されていない場合は、このアクセスが競合していないとわかるため、そのアクセスに関する処理を続行する。一方で、ロックが獲得されていた場合、以下の状況が考えられる。

- (1) この Tx 内で当該変数に対してすでに Write アクセスしたため、**競合していない**。
- (2) 他の Tx が当該変数に対して Write アクセスしたため、**競合している**。

どちらの場合かを判別するために、当該 Tx の Write-set に当該変数が記録されているかを確認する。記録されていた

Time	Thr.0	Thr.1
t0	a = 0; b = 0; x = 0; y = 0;	
t1	TX_BEGIN(0);	
t2	tmp_x = x;	
t3	tmp_y = y;	
t4	tmp_a = a;	TX_BEGIN(1);
t5		a = 1;
t6		b = 1;
t7		TX_END(1);
t8	tmp_b = b;	
t9	TX_END(0);	

図 2: TinySTM における Opacity 保証の工夫が効果的な例

場合、その Tx がすでに当該変数に Write アクセスし、ロックを獲得したとわかるため、当該アクセス処理を完了する。一方で、Write-set に記録されていなかった場合、他の Tx がすでに Write アクセスしており、かつその Tx がまだコミットしていない状況であるため、このアクセスが競合しているとわかる。この競合を解決するために、TinySTM では、ロックが獲得されている変数にアクセスした Tx を常にアボートするようにしている。

TinySTM では、共有変数ごとにロックを用意して競合を検出するため、競合アクセス以外のアクセスを制限することがなく、競合が少ない場合に高い並列性を発揮しうる。

3.1.2 Opacity 保証

TinySTM では Opacity 保証のため、共有変数への Read アクセス時またはコミット直前において、それまでに Read したすべての共有変数が更新されていないかを検証する。この動作のうち、Read アクセスごとに行うものを **Incremental Validation** [7] という。ここで、共有変数への更新を確認するため、共有変数ごとにバージョン番号を管理する。バージョン番号は、紐付いた共有変数に Write アクセスした Tx（以降、**Writer** という）のコミット時に更新される。Incremental Validation における共有変数の検証は、共有変数へ Read アクセスする Tx（以降、**Reader** という）が共有変数へ Read アクセスするたびに実行される。これは、共有変数への Read アクセス時に当該変数のその時点でのバージョン番号を Read-set に記録しておき、この値と検証時の値とを比較することで実現される。

TinySTM では、Incremental Validation の回数を削減する工夫が施されている。具体的には、共有変数への Read アクセスごとに Incremental Validation の必要性を軽量に検知し、必要な場合にのみ実行する。この仕組みは、システム全体で共有される単一のカウンタを用いて実現され

ている。この共有カウンタは、Writer がコミットする際にアトミックにインクリメントされる。そして、インクリメント後の値は Writer が Write アクセスした共有変数のバージョン番号として記録される。また、Tx は実行を開始したとき、その時点における共有カウンタの値を保存する。Tx が共有変数に Read アクセスしたとき、当該変数のバージョン番号とはじめに保存したカウンタの値とを比較し、前者の値が後者の値以下であれば、それまでの Tx 実行ステップにおいて実行結果が一貫していることがわかるため、Incremental Validation をしなくても問題ないといわれる。このようにして、TinySTM では Incremental Validation の回数を削減している。

この仕組みを踏まえた Opacity 保証の流れについて、図 2 に示す具体例を用いて説明する。ここで、共有カウンタの初期値が 0 であり、共有変数 a, b, x, y のバージョン番号も 0 であったとする (t0)。Thr.0 が Tx の実行を開始したとき、開始時の共有カウンタの値として 0 を保存する (t1)。次に、Thr.0 が x を Read したとき、そのバージョン番号が 0、実行開始時の共有カウンタの値も 0 であり、 $0 \leq 0$ が成り立つため、Incremental Validation を省略する (t2)。また、y, a への Read アクセスについても同様であり、Incremental Validation を省略する (t3, t4)。さらに、Thr.1 が Tx の実行を開始する (t4)。ここでも、Write アクセスをする他の Tx がコミットしていないため、実行開始時の共有カウンタの値として 0 を保存する。続いて、Thr.1 が a, b に対して Write アクセスしたのちコミットするとき、共有カウンタの値を 1 に変更し、a, b のバージョン番号も 1 に変更する (t7)。そして、Thr.0 が b に Read アクセスしたとき、b のバージョン番号が 1 であるとわかり、 $1 \leq 0$ は成り立たないため、Incremental Validation を実行する (t8)。そこで、x, y は更新されていないものの、a へ Read アクセスしたのちに他の Tx によって a が更新されていることを検知できるため、Opacity 保証のために Thr.0 はアボートする。仮に TinySTM における Opacity 保証の工夫がなかった場合、時刻 t3 においては共有変数 x について、時刻 t4 においては共有変数 x, y についての Incremental Validation が必要となっていた。

3.1.3 問題点

TinySTM の問題点として、Incremental Validation の回数を削減する工夫はしているものの、Incremental Validation のコストが大きいということが挙げられる。Tx 実行における i 番目の Read アクセスでは、それまでに Read アクセスした $(i - 1)$ 個の共有変数について更新の有無を確認する必要がある。したがって、共有カウンタを用いても最悪時は Read アクセスごとに Incremental Validation が必要になる。このとき、ある Tx における Read アクセスの合計数を N とすると、変数に対する更新の有無を確認する回数の合計が $\sum_{i=1}^N (i - 1) = \frac{N}{2} (N - 1)$ となる。その

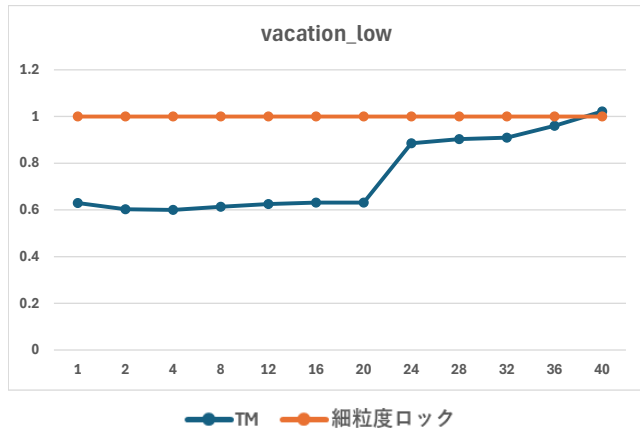


図 3: TinySTM と細粒度ロックのベンチマーク評価

表 1: 評価環境

OS	Ubuntu 20.04
Processor	Intel Xeon Platinum 8380
clock	2.30 GHz
cores	40 cores
L1-dcache	1.9 MiBytes
L2-cache	50 MiBytes
L3-cache	60 MiBytes
Memory	64 GiBytes
Compiler	gcc version 9.4.0
Compile options	-O2

ため、Tx 実行を通して Incremental Validation のコストが $O(N^2)$ となってしまふ。これにより、LBP 実装のものよりも、TinySTM が低い性能を示すワークロードが存在する。

図 3 は、TM の性能を評価するベンチマークである PANDORA (Parallel-and-Distributed-computing-Oriented Redesigned Application program suite) [8] に含まれる、vacation_low を用いて TinySTM と細粒度ロックとの性能を比較した結果を示している。評価環境を表 1 に示す。1 コアあたり 4 スレッドを生成する設定でプログラムを実行した。縦軸は細粒度ロックを基準とした実行速度比、横軸はプログラムの実行コア数を示している。これらのプログラムは、PANDORA に備わっている他のプログラムと比べて、1 つの Tx が Read アクセスする共有変数の個数が多い。そのため、Incremental Validation のコストが大きく、図 3 で細粒度ロックと比べて TinySTM が低い性能を示していたと考えられる。

3.2 InvalSTM

3.2.1 Invalidation

InvalSTM では、TinySTM とは異なり、無効化 (Invalidation) 方式で共有変数の整合性を保つ。TinySTM では、Reader 主体の動作によって整合性が保たれていた。

具体的には、共有変数ごとのロックの確認による競合検出と、Incremental Validation による Opacity 保証である。一方で、無効化方式では、コミット直前の Writer が、並列に実行中である他の Tx を検証することで、整合性を保つ。具体的には、競合アクセスをした Tx や、Opacity を保証できない Tx が、Writer によってアボートさせられることで実現される。

無効化方式では、Tx それぞれに用意された valid フラグという変数を False に変更することによって、不整合状態の Tx をアボートさせる。なお、競合検出時にアボートさせる対象となるのは、検証している Tx または実行中の (in-flight な) Tx のいずれかである。

TinySTM では、Opacity 保証のための Incremental Validation のコストが大きかったのに対して、無効化方式では、Read アクセスごとの Opacity 保証コストが小さいという利点がある。無効化方式における Opacity 保証は、Tx が共有変数へ Read アクセスすることに自身の valid フラグを確認して、False になっていた場合にアボートする処理を行うことで実現される。Tx 実行における i 番目のアクセスでは、その valid フラグを 1 回確認する必要があるため、ある Tx における Read アクセスの合計数を N としたとき、Read アクセスに併せて必要となる Opacity 保証コストは $O(N)$ となる。

無効化が正しく実行されるようにするため、InvalSTM ではロックを用いて Tx の実行を制御している。InvalSTM におけるロックを用いた制御は、TinySTM と比較して用いられるロックの粒度が異なる。InvalSTM では、共有変数ごとにロックが用意されているのではなく、コミットロック、in-flight ロックという 2 種類のグローバルなロックと、Tx ロックという Tx ごとに固有のロックが用意されている。

3.2.1.1 コミットロック

このロックは、検証を始めようとする Tx によって獲得され、その Tx の実行が終了する際に解放される。このロックにより、同時に検証を実行できる Tx の数が 1 つに制限される。仮に検証が並列に実行できてしまうと、ある検証 Tx が他の検証 Tx すべてに対する検証を完了することを保証できない。

3.2.1.2 in-flight ロック

このロックは、検証を始めようとする Tx のほか、検証対象となる実行中の Tx を管理するデータ構造 (以降、in-flight-set と呼ぶことがある) が更新される際に獲得される。また、検証する Tx が終了する際、もしくは in-flight-set の更新動作が完了した際に解放される。このロックにより、検証中に新規の Tx が実行を開始できなくなる。仮に検証中に新規の Tx が開始できてしまうと、検証する Tx がその Tx を検知できない。そのため、その Tx による競合アクセスが発生した場合に Serializability が毀損されて

Time	Thr.0	Thr.1	Thr.2
t0	a = 0; b = 0;		
t1	TX_BEGIN(0);		
t2	tmp_a = a;		TX_BEGIN(2);
t3			a = 2;
t4		TX_BEGIN(1);	
t5		a = 1;	
t6		b = 1;	
t7	tmp_b = b;	TX_END(1);	b = 2;
t8	TX_END(0);		TX_END(2);

図 4: 複数の Tx が競合する例

しまう。

3.2.1.3 Tx ロック

このロックは、検証を始めようとする Tx のほか、このロックが用意されている Tx の Read-set および Write-set が更新される際に獲得される。また、検証する Tx が終了する際、もしくは当該 Read-set および Write-set の更新が完了した際に解放される。このロックにより、検証中、検証対象となっている Tx が新たに共有変数へアクセスできなくなる。仮に上述のようなアクセスができてしまうと、当該 Tx に対する検証が完了したあとの当該 Tx による共有変数へのアクセスが検証されない。そのため、競合アクセスが検知できず、Serializability が毀損されうる。

無効化による競合検出、Opacity 保証の動作を、図 4 に示す具体的なコードを用いて説明する。ここで、ここに登場するスレッド以外は動作していないとする。まず、Thr.0 が Tx 実行を開始する。Thr.0 は、in-flight ロックを獲得し、in-flight-set に自身を登録したのち、in-flight ロックを解放する (t1)。続けて、自身の Tx ロックを獲得し、共有変数 a への Read アクセスを行う。このとき、共有変数 a への Read アクセスを行った Thr.0 は自身が実行している Tx の valid フラグを確認するが、その時点ではいずれの Tx からその valid フラグを変更されておらず、True であるため、Tx 実行を継続する。Read-set の更新など Read アクセスの処理が終わると、Tx ロックを解放する (t2)。また、Thr.2 が Tx 実行を開始する (t2)。ここで行う処理は Thr.0 のときと同様である。続けて、Thr.2 は a へ Write アクセスする。このとき、Thr.0 の Read アクセスのときと同様に、Write アクセス自体の処理のほか、Tx ロックの獲得、解放を行う (t3)。次に、Thr.1 が Tx 実行を開始する (t4)。Thr.1 は、共有変数 a, b への Write アクセスを行い (t5-t6)、他の Tx を検証するフェーズに入る (t7)。ここで Thr.1 は、はじめにコミットロック、in-flight ロック、Thr.0, 1, 2 の Tx ロックを獲得する。このとき、以下の状況にあるため、いずれのロック獲得も失敗しない。

- 他の Tx がコミットフェーズに入ろうとしていない。
- 新規の Tx が実行を開始しようとしていない。

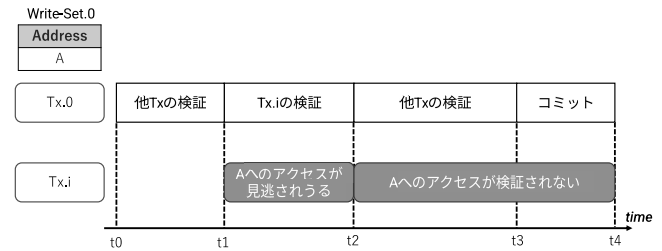


図 5: 無効化の並列実行に必要な制御

- 各 Tx が新たに共有変数にアクセスしようとしていない。

このため、Thr.1 はその時点で並行して Tx を実行している Thr.0, Thr.2 の検証を行うことができる。検証では Thr.0 で実行されている Tx の Read-set を確認し、自身が Write アクセスした a がすでに Read アクセスされており WaR 競合が発生していることがわかるため、Thr.0 で実行されている Tx の valid フラグを False に変更する。また、Thr.2 の検証では、その Write-set に a が登録されており、WaW 競合が発生していることがわかるため、Thr.2 の valid フラグを False に変更する。これで、Thr.1 は並列動作するすべての Tx の検証を終えたため、当該 Tx をコミットする。

3.2.2 問題点

InvalidSTM の問題点は、検証中に他の Tx の動作を過度に制限しており、並列性を毀損してしまっているという点にある。これについて、検証中において共有変数に対してどのようなアクセス制御が必要なのか、図 5 を用いて説明する。図 5 は、Tx 内における Read アクセス、Write アクセスを終え、コミットするために検証を始める Tx.0 と、この時点で並列実行している Tx.i の様子を示している。また、Tx.0 は共有変数 A への Write アクセスを行ったため、その Write-set に A が記録されている。Tx.0 による検証が開始してから Tx.i の検証が始まるまでの間 (t0-t1)、Tx.i によって A へのアクセスが実行された場合、Tx.0 は時刻 t1 から始まる Tx.i の検証においてこのアクセスを検出できる。一方で、Tx.i の検証が行われている間 (t1-t2) に Tx.i によって A へのアクセスが実行された場合、Tx.0 による Tx.i の Read-set, Write-set の走査の進行具合によってはそのアクセスに対する検出が漏れてしまう場合がある。このとき、その他に Tx.i について競合が検出されず、Tx.i がアボートしなかった場合、競合しており不整合状態にある Tx がコミットしてしまうこととなる。また、Tx.i の検証が終了してから Tx.0 のコミットが完了するまでの間 (t2-t4)、アボートせずに実行を継続している Tx.i が A へアクセスした場合、Tx.0 による Tx.i の検証は完了してしまっているため、この競合に関して Tx.i がアボートすることはない。この場合、同様に不整合状態にある Tx がコミットしうるため問題である。InvalidSTM では、時刻 t0 の時点で、コミットロック、in-flight ロック、並列動作

している Tx すべての Tx ロックを獲得しているため、時刻 t_0 から時刻 t_4 までの間、Tx.i は新たに共有変数に対するアクセスを行うことを禁止されるため、この問題が発生することはない。

ここで、例えば Tx.0 によって Write アクセスされていない共有変数 B があったとする。このとき、時刻 t_0 から t_4 の間に Tx.i が B に対してアクセスする場合、これは競合しておらず、問題を引き起こさない。しかし、InvalidSTM ではこの間、いかなる共有変数へのアクセスも禁止されているため、B へのアクセスも当然実行できない。

このように、無効化方式では検証する Tx (図5では Tx.0) の Write-set に含まれる共有変数について、その検証が開始してからコミットが完了するまでに行われるアクセスを禁止すればよいが、InvalidSTM では禁止する必要のないアクセスについても禁止しているため、並列性を損なっている。

4. 提案手法

本章では、3章における既存手法の分析を踏まえて、軽量の一貫性保証と高い並列性を両立する TM を提案する。

4.1 既存実装の課題

3章で説明した TinySTM, InvalidSTM は、それぞれ利点と欠点を抱えていた。これらの STM 実装と、これから設計していく提案手法について、並列性、Read アクセス時に必要な Opacity 保証コストという観点から特性をまとめたものを表2に示す。

TinySTM は共有変数ごとに用意したロックを用いて競合検出を行っていた。そのため、実際に競合するアクセスが少ない場合、一貫して Tx が並列動作し、並列性が高まりやすい。しかし、Opacity 保証のためにコストの大きい Incremental Validation をする必要があり、Read-set が大きくなるワークロードでは高い性能を発揮できないことがあった。一方、InvalidSTM は Opacity 保証を無効化方式で実現するため、Read アクセスが多いワークロードでも Opacity 保証コストが増加しづらいという利点がある。しかし、TinySTM とは異なり、Tx の制御に共有変数ごとのロックを用いず、Tx ロックなどを用いた粗粒度ロックを採用しているため、アクセスが競合していなくとも単一 Tx の逐次実行が行われ、並列性を毀損してしまっている。

4.2 設計方針

競合検出すなわち並列性の確保と、一貫性 (opacity) 保証を分離することが、提案手法の重要な方針である。本節では、TinySTM と InvalidSTM の利点を矛盾なく統合する設計方針について述べる。

細粒度ロックによる並列性の最大化

提案手法では、TinySTM のように共有変数ごとにロック

を用意し、これを用いて共有変数へのアクセス調停を行う。このロックは、それに対応する共有変数に対する Write アクセスリクエストが発生した際に獲得される。Tx は共有変数にアクセスする際、はじめにその変数に対応するロックを確認する。そのロックが獲得されていなければ競合していないため、アクセス処理を続行する。一方、ロックが獲得されていた場合は競合しているため、アボートする。獲得されたロックは、ロックを獲得した Tx がコミットした場合とアボートした場合に解放される。

無効化方式による一貫性保証の軽量化

Opacity 保証は、InvalidSTM と同じく無効化方式を採用する。Tx は共有変数への Read アクセス時に、当該変数のロックが獲得されていないことを確認するだけでなく、自身が Invalidate されていないかの確認も行い、Invalidate されていた場合はアボートする。また、検証を完了したのち、コミットする直前にも同様のチェックを実行する。これにかかるコストは、3.2.1 項で説明したように $O(N)$ となる。また、1つ以上の共有変数に Write アクセスする Tx は、コミット処理を開始する前に並列動作する Tx の Read-set を検証する。そこで、自身が Write アクセスした変数が含まれていることを発見した場合、自身もしくは検証対象の Tx のいずれかをアボートする。

一方で、共有変数ごとのロックを用いて Tx を制御するため、InvalidSTM のような粗粒度なロックは採用しない。3.2.2 項で説明したように、無効化方式における Tx の制御として、InvalidSTM は Tx の並列動作を過剰に制限している。しかし、提案手法では、Writer は書き込み対象の変数ロックを「検証開始前」に全て獲得し、コミット完了まで保持し続ける。これにより、検証中に他の Tx が同一変数へ書き込む、あるいは新たに Read-set へ追加して見逃されるといった、直列化可能性を脅かす競合状態を排除できる。また4.3節で詳述するように、Tx のディスクリプタ管理に CAS 命令を活用する。これにより、実行を開始したものの TxArray の更新に失敗し、検証を見逃される Tx が生じることによる検証漏れを防ぐことができる。

この設計により、Serializability と Opacity を保証することができる。Writer が Write アクセスする共有変数のロックを獲得するため、WaW 競合と RaW 競合は検出可能である。また、検証漏れが防がれており、かつコミット直前の Tx が無効化確認を行うため、WaR 競合となる Tx がいづれもコミットすることはなく、WaR 競合も検出可能である。したがって、Serializability は保証される。続いて、Tx は Read アクセスごとに自身の無効化確認を実行し、また Writer は他の全 Tx を検証と自らの無効化も確認を完了した後コミットするため、Opacity を毀損するような共有変数の Read および Tx のコミットは行われぬ。そのため、Opacity も保証される。

表 2: 既存手法および提案手法の特性

観点	TinySTM	InvalSTM	提案手法
並列性	競合していない場合は 並列実行可能であり 並列性が高い	競合していない場合も 逐次実行部分があり 並列性が低い	競合していない場合は 並列実行可能であり 並列性が高い
Read アクセス時に必要な Opacity 保証コスト	$O(N^2)$	$O(N)$	$O(N)$

4.3 実装

性能の鍵となる，実行中 Tx の管理，および，Read-set 表現の詳細について述べる．

実行中 T_x の管理構造

無効化方式では、検証を行う Tx が検証対象のスレッドを把握するために、システム全体で共有される、実行中の Tx を示すリストのようなデータ構造が必要である。そのため、グローバルな領域に、実行中の Tx のディスクリプタを格納するデータ構造を用意する。

このデータ構造として複数の実装を試したが、ワークロードを実行するコア数分の要素をもつ配列が最も高い性能を示したため、これを採用した。固定長配列は、プロセッサのキャッシュライン活用およびメモリアクセスの予測可能性が最大化できるという点でメリットがある。以下これを TxArray と呼ぶ。

新たに Tx を開始する際には、TxArray を操作し、アクティブでない要素に自身を登録する。この際、複数のスレッドが同一要素に対して競合することを防ぐため、Compare-and-Swap (CAS) 命令を用いた以下のようなアルゴリズムを採用した。この手順により、検証対象となる Tx の集合を常に正確に維持することが可能となる。

- (1) TxArray を走査し、アクティブでない要素 i を特定する
- (2) CAS 命令により、要素 i の状態をアクティブにアトミックに変更することを試みる
- (3) CAS 命令が成功した場合は登録完了となる．一方で失敗した場合は、要素の探索を継続する．

Bloom filter による Read-set の高速検証

無効化方式では、検証する Tx が他の並列実行中の Tx の Read-set を検証する必要がある。この検証では、検証 Tx の Write-set に含まれる共有変数が、検証対象の Tx の Read-set に含まれているかを確認する。ここで、単純に配列などで Read-set を実装した場合、1 つの共有変数についての検証を完了するには検証対象の Tx の Read-set 全体を走査する必要がある。

このコストを低減するため、InvalSTM にも採用されている、Tx の Read-set を Bloom filter [9] で実装する手法を導入する。Tx は固有のビット配列を有しており、これを Read-set として用いる。Tx は共有変数に Read アクセスする際、その変数のアドレスからハッシュ値を計算し、

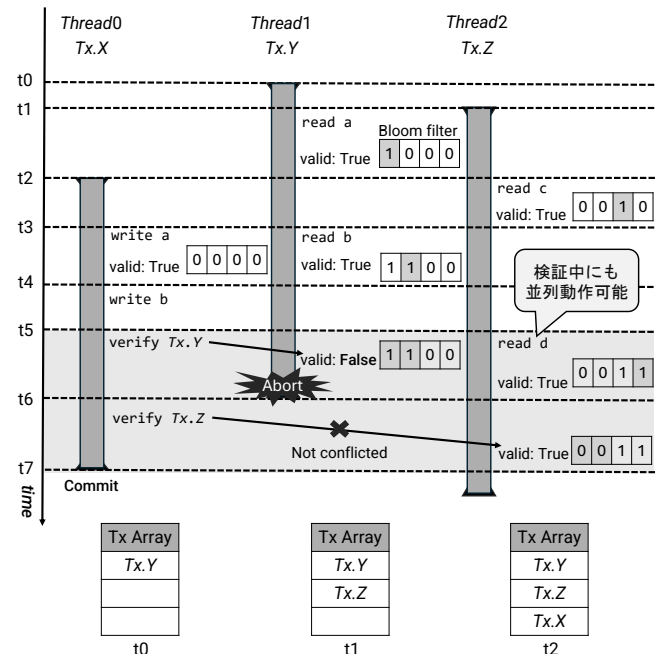


図 6: 提案手法の動作例

ビット配列の対応するインデックスの要素を 1 に変更することで, Writer からこの Read アクセスが検知されるようにする. Bloom filter における要素の検索は, 常に一定量のハッシュ計算で完結するため, Tx の Read-set に含まれる要素数が多いワークロードでは, 特に Bloom filter を利用する利点大きい.

なお、Bloom filter の特性上、ハッシュ衝突による偽陽性 (false positive) が発生し、競合していない Tx を誤って無効化する可能性がある。しかし、これは不必要なアボートを招くのみで、Serializability を損なうことはないため、正当性は担保される。また、Read-set のサイズが大きくなり、ビット配列のサイズが相対的に十分でなくなると、偽陽性の発生が増加し、アボート率が上昇する可能性がある。そのため、Bloom filter に用いるビット配列のサイズは十分に大きく設定した。

4.4 動作

図 6 を用いて，設計した提案手法の動作を説明する．ここで，図 6 に示されているスレッドのみが動作しているとす．まず，*Thread1* が *T_x.Y* の実行を開始する (*t*₀)．このとき，*Thread1* は *T_x* 配列に対し，新規に実行を開始した

T_x のディスクリプタを追加する．次に， $Thread1$ が共有変数 a へ Read アクセスする ($t1$)．ここで， a のロックは獲得されていないため，この Read アクセス処理は継続する． $Thread1$ は共有変数へ Read アクセスしたため，自身の Bloom filter に対して当該変数に対応するインデックスを 1 に変更する． $T_x.Y$ は他のスレッドから Invalidate されていないため，その valid フラグは初期状態の True のままである．また，同時に $Thread2$ が新規に T_x の実行を開始する．ここでも，新規に実行した T_x のディスクリプタを T_x 配列に追加する．続いて， $Thread2$ が共有変数 c へ Read アクセスする ($t2$)．このアクセスも， c のロックは獲得されていないため，続行する．このとき， $Thread2$ は c に対応するインデックスについて，自身の Bloom filter の要素を 1 に変更する．また， $Thread0$ が $T_x.X$ の実行を開始する．その後， $Thread0$ が a へ Write アクセスする ($t3$)．このアクセスは Write アクセスであるため， $Thread0$ は a を自身の Write-set に追加するが，Read-set である Bloom filter は変更しない．さらに， $Thread0$ は a のロックを獲得する．また， $Thread1$ が共有変数 b に Read アクセスする．同様に， $Thread1$ は自身の Bloom filter における b に対応するインデックスの要素を 1 に変更する．そして， $Thread0$ は b に Write アクセスする ($t4$)．同様に，Write アクセスであるため， $Thread0$ は b のロックを獲得する．この Write アクセス完了後， $Thread0$ はコミットフェーズに入り，並列動作する T_x の検証を開始する．まず， $Thread0$ は T_x 配列のはじめに登録されている $T_x.Y$ について検証する ($t5$)．ここで， $Thread0$ は $T_x.Y$ の Bloom filter に対して， a, b に対応するインデックスの要素を確認する．そこで， $Thread0$ はその要素が 1 になっていることを検出するため，これを WaR 競合だと判定し， $T_x.Y$ の valid フラグを False に変更する．これにより，valid フラグが False となった $T_x.Y$ はアボートすることとなる．また， $Thread2$ が共有変数 d への Read アクセスを実行する．このとき， $Thread0$ が検証を実行中であるものの， d のロックは獲得されていないため，このアクセスは許可される ($t5$)．次に， $T_x.Y$ の検証を完了した $Thread0$ は， T_x 配列を参照し， $T_x.Z$ を発見するため，これについての検証を開始する ($t6$)． $T_x.Z$ の Bloom filter は， $Thread0$ が Write アクセスした共有変数 a, b に対応する要素が 1 となっていない．そのため， $Thread2$ は競合アクセスをしておらず， $T_x.Z$ の valid フラグは True のまま変更されない． T_x 配列には $T_x.X$ があるが，これは検証 T_x そのものであるため，検証を完了する．検証を完了した $Thread0$ は実行している $T_x.X$ の valid フラグを確認し，True であることがわかるため， $T_x.X$ はコミットする ($t7$)．

5. 評価と考察

本章では，提案手法を評価する．

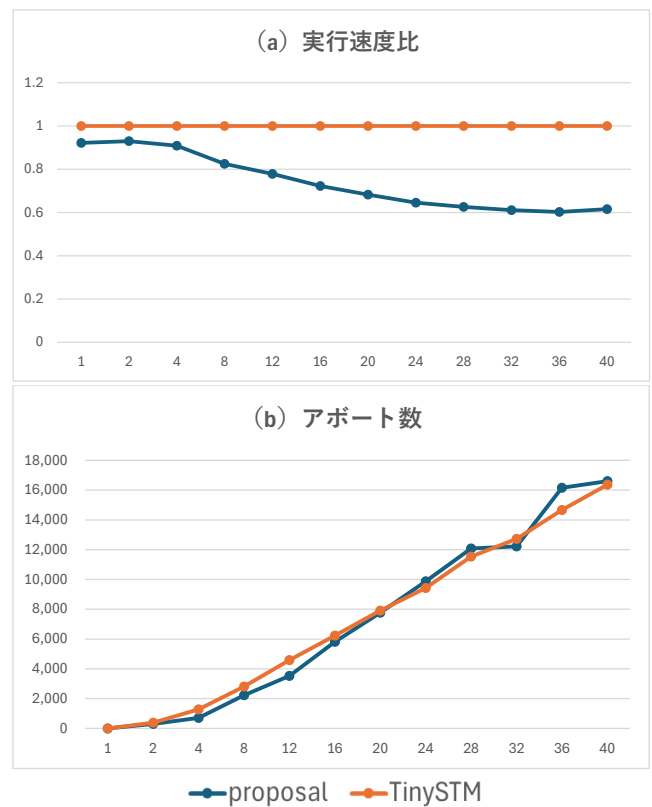


図 7: vacation_low における提案手法の評価結果

5.1 評価結果

評価環境は 3.1.3 項での評価と同様で，表 1 に示すとおりである．評価には，PANDORA に含まれる様々なベンチマークプログラムを用いた．なお，ベンチマークでは，コア 1 つにつき 4 スレッドを生成するように設定した．この評価では，提案手法との比較対象として，提案手法の実装のベースとした TinySTM を用いた．

図 7 に vacation_low を用いた評価結果を示す．図 7.(a) のグラフは TinySTM を基準とした実行速度比，図 7.(b) のグラフはアボート数を示している．また，横軸はどちらのグラフもベンチマークの実行コア数を示している．

図 7.(a) より，vacation_low において提案手法は，TinySTM と比較して低い性能を示した．また，実行コア数が増加していくにつれて，提案手法の実行速度比は低下した．提案手法の実行速度比について，最大となったのは，2 コア実行時で 0.93 であった．一方で，最小となったのは，36 コア実行時で 0.61 であった．続いて，図 7.(b) のグラフが示す結果より，実行コア数がいずれの場合でも提案手法と TinySTM とではアボート数に大きな差を示さなかった．

次に，図 7 と同様の形態で図 8 に kmeans_low を用いた評価結果を示す．

図 8.(a) より，kmeans_low では，提案手法は TinySTM と比較して，実行コア数が 24 以下のときに実行速度に差をほとんど示さなかった．しかし，実行コア数がそれより

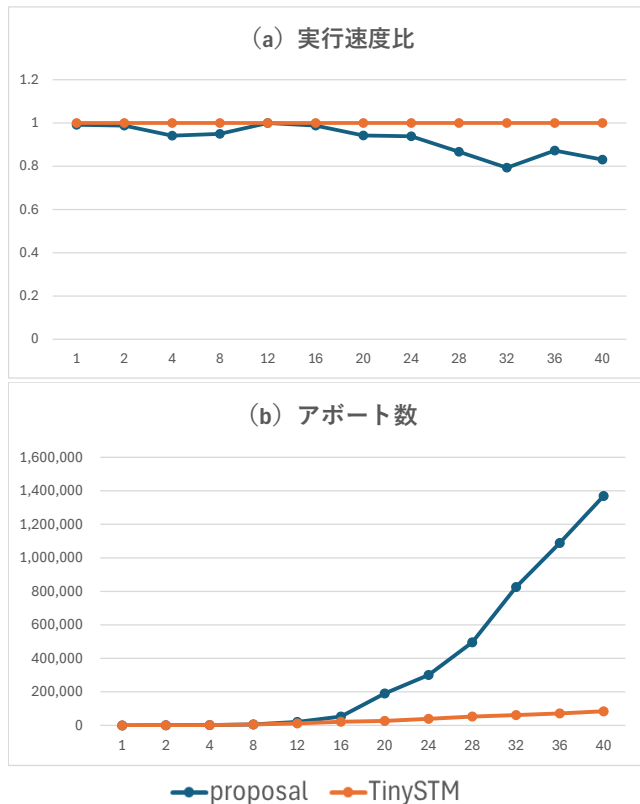


図 8: kmeans_low における提案手法の評価結果

多くなると、提案手法は TinySTM を下回った。提案手法の実行速度比について、最大となったのは、12 コア実行時で 1.0 であった。一方で、最小となったのは、32 コア実行時で 0.79 であった。また、図 8.(b) のグラフが示す結果より、実行コア数が 20 以上の場合は、提案手法のアボート数は TinySTM のそれと比べて多い。

5.2 考察

評価結果から、現状の提案手法は TinySTM と比べて性能向上を達成するには至らなかった。しかし本評価結果は、無効化方式を細粒度ロック環境に導入した際に支配的となるボトルネックを定量的に特定するものであり、高性能 STM 設計に向けた実装指針を与える重要な知見である。以下、3 つの観点から考察する。

検証コストのスケラビリティ

図 7 に示される vacation_low での評価では、アボート数においては提案手法と TinySTM に大きな差を示さなかった。しかし、実行速度について、実行コア数が増加するにつれて提案手法は TinySTM を下回る。ここで、コア数の増加につれて実行速度が低下してしまう原因を分析するため、vacation_low の 40 コアでの実行をプロファイリングした。その結果の一部を表 3 に示す。なお、stm とつかない関数はベンチマーク固有の関数であり、stm とつく関数は TM システムで用いられる関数である。そのうち、いくつかの関数についてどのような関数なのか説明する。

- stm_load.tx: 共有変数への Read アクセス時に呼ばれる関数
- stm_store.tx: 共有変数への Write アクセス時に呼ばれる関数
- stm_commit.tx: Tx がコミットする際に呼ばれる関数
- stm_verify: Writer によるコミット前検証の際に呼ばれる関数
- stm_add_to_r_set_bloom: 共有変数への Read アクセス時に Bloom filter を更新する関数
- stm_enter.tx: Tx 開始時に TxArray を更新する際に呼ばれる関数

表 3 から、提案手法では、Writer がコミット前に行う検証 (stm_verify) が大きなオーバーヘッドとなってしまうことがわかる。コミット前検証では、Writer はコミット前に並列に Tx を実行しているスレッドをすべて検証する必要があり、特に実行コア数が多い場合に、このコストが理論上の一貫性保証の軽量化による恩恵を相殺してしまうことで、性能向上の制約要因となっている。一方で、実行中の Tx を管理する TxArray に関する操作 (stm_enter.tx) が実行時間を占める割合は小さいことから、スレッド管理機構自体の実装は十分に効率的であることも確認できた。よって今後、検証フェーズの並列化や検証対象の絞り込みによって、検証コストを軽量化することが必要である。

Bloom filter による Read-set 管理の有効性と課題

stm_verify に加え、Tx が Read アクセスごとに行う Bloom filter に関する操作 (stm_add_to_r_set_bloom) も大きなオーバーヘッドとなっていた。特に、Read アクセスの多い vacation_low において、stm_add_to_r_set_bloom は大きな割合を占めた。

ここで、図 7.(a) において、実行コア数が多くない状況でも提案手法の実行速度が TinySTM のそれを下回っている原因を分析するため、vacation_low の 4 コアでの実行において、プロファイリングを実行した。その結果を表 4 に示す。

表 4 から、4 コア実行においても、提案手法では各 Read アクセスにおける Bloom filter に関する操作 (stm_add_to_r_set_bloom) が大きなオーバーヘッドとなっていることがわかる。一方で、Incremental Validation を軽量化したかわりに必要となったコミット前検証 (stm_verify) にかかるコストは小さいということもわかった。

この結果を受けて、Bloom filter を利用せず、共有変数の情報をもつ構造体配列である、従来の Read-set を用いるように提案手法の実装を変更したものと、実装を変更していないものとで性能比較を実施した。その結果を図 9 に示す。

図 9 から、従来の Read-set を用いる実装では、vacation_low での多コア実行において、実行速度が低下していることがわかる。これは、vacation_low は共有変数への

表 3: vacation_low の 40 コア実行でのプロファイリング結果

提案手法		TinySTM	
Overhead(%)	Symbol	Overhead(%)	Symbol
31.60	stm_load_tx	48.08	stm_load_tx
17.20	stm_verify	9.55	stm_commit_tx
17.12	stm_add_to_r_set_bloom	8.38	rbtree_get
5.98	rbtree_get	4.93	TMlookup
3.11	TMlookup	4.15	stm_start_id
2.30	stm_commit_tx	2.04	TMfindPrevious
...	...	1.92	reservation_info_compare
1.65	stm_enter_tx	1.82	stm_store_tx

表 4: vacation_low の 4 コア実行でのプロファイリング結果

提案手法		TinySTM	
Overhead(%)	Symbol	Overhead(%)	Symbol
46.21	stm_load_tx	53.14	stm_load_tx
11.52	stm_add_to_r_set_bloom	9.41	rbtree_get
8.59	rbtree_get	6.01	stm_commit_tx
4.45	TMlookup	5.66	TMlookup
2.30	stm_verify	2.11	TMfindPrevious
...	...	1.88	reservation_info_compare
0.72	stm_enter_tx	1.59	fixAfterDeletion

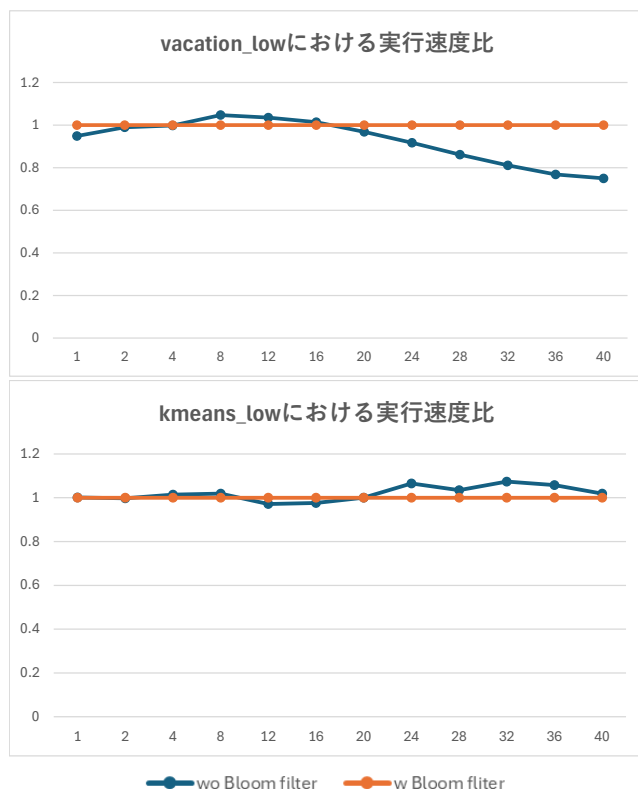


図 9: Bloom filter の有無による実行速度比較

Read アクセスが多く、Read-set のサイズが大きくなるため、検証における検証対象の Tx の Read-set 走査に時間が長くなるためだと考えられる。一方で、kmeans_low においては、多コア実行においては若干の性能向上が見られる

ものの、Bloom filter の有無による性能差があまりないことがわかる。性能差が小さいのは、kmeans_low は共有変数へのアクセスが少なく、Read-set のサイズが大きくなりづらいため、Bloom filter による Read-set 検索の高速化の影響が小さいためだと考えられる。また、若干の性能向上が見られたことから、Bloom filter を利用することによる利点が小さい状況では、Bloom filter 導入によるハッシュ計算などのコストが支配的となることで、性能劣化を招くと考えられる。

これらは、従来のリスト走査コストを定数時間 $O(1)$ に抑える代償として、ハッシュ計算やメモリ書き込みのオーバーヘッドが無視できないことを示している。しかし、これは Read-set が大きくなるワークロードにおいては逆転することがあることも示しており、一貫性保証コストの増加を抑制するためのデータ構造の選択において、重要な定量的知見であると言える。

ロック保持期間とアバート率との関係

図 8 での kmeans_low による評価において提案手法では、実行コア数が増加すると、アバート数が顕著に増加している。これも、実行コア数の増加による検証コストの増加が原因であると考えられる。一方で、vacation_low での評価結果と異なり、顕著なアバート数の増加がみられたのは、kmeans_low では Tx 1 回あたりの実行にかかる時間が比較的短いためだと考えられる。提案手法では、共有変数に対して Write アクセスしてから検証が終了しコミットするまでの間、当該変数に関するロックを獲得する。このとき、

検証対象のスレッド数が増大し、検証時間が長くなると、変数のロックが保有される時間も長くなる。これが後続するTxによるアクセスを阻害することで、不必要なアボートを誘発している。そのため、Tx 1回あたりの実行時間が短く、TinySTMのような実装では変数のロックが保有される時間が短いワークロードでは、提案手法だと、Txがアクセスしようとする変数のロックが獲得されていることによるアボートが頻発すると考えられる。

6. おわりに

本稿では、従来のSTM実装が抱える一貫性保証コストと並列性とのトレードオフについて分析し、それらを解消する新たな設計を提案した。具体的には、細粒度ロックによる高い並列性を維持しつつ、無効化方式を導入することで $O(N)$ のOpacity保証コストを実現するSTMを設計・実装した。

評価の結果、実装した提案手法は多くの場合TinySTMの性能を下回る結果となった。しかし、詳細なプロファイリングを通じて、多コア環境下におけるコミット前検証コストの増大や、Bloom filter操作に伴うハッシュ計算等のオーバーヘッドが、理論上の計算量削減による恩恵を相殺してしまうという、実効性能上の境界条件を明確に特定した。これは、高並列なSTMの実現において、実装に伴う実効的なオーバーヘッドを極限まで抑える実装技術が極めて重要であることを浮き彫りにした。

今後の展望として、実装した提案手法のオーバーヘッドについての調査を進め、性能を向上させていくことが考えられる。現時点で考えられる改善点として、Tx内のReadアクセス数に応じてBloom filterの利用を動的に切り替える仕組みや、検証対象を絞り込んで検証時間を短縮する手法の導入が挙げられる。これらの改善を通じて、提案した設計が持つ本来のポテンシャルを引き出し、実用的なSTMの実現へと繋げていきたい。

謝辞 本研究の一部はJSPS科研費21H03408, 23K21652, 25K03093の助成を受けたものである。

参考文献

- [1] Herlihy, M. and Moss, J. E. B.: Transactional memory: Architectural support for lock-free data structures, *Proceedings of the 20th annual international symposium on Computer architecture*, 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] Shavit, N. et al.: Software Transactional Memory, *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pp. 204–213 (1995).
- [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] Gottschlich, J. E., Vachharajani, M. and Siek, J. G.: An efficient software transactional memory using commit-time invalidation, *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 101–110 (2010).
- [6] 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).
- [7] Harris, T., Larus, J. and Rajwar, R.: *Transactional Memory, 2nd Edition*, Morgan and Claypool Publishers (2010).
- [8] 佐藤宏樹, 酒井駿輔, 伊原 楓, 小泉 透, 塩谷亮太, 五島正裕, 津邑公暁: トランザクショナルメモリの性能を正確に評価可能なベンチマークスイートの開発, *The 9th cross-disciplinary workshop on computing Systems, Infrastructures, and programming (xSIG 2025)* (2025).
- [9] Bloom, B. H.: Space/time trade-offs in hash coding with allowable errors, *Communications of the ACM*, Vol. 13, No. 7, pp. 422–426 (1970).