環境情報学部論文

並行性制御法におけるROS TFの高品質化

2022年1月

71970013 / t19501yo
荻原　湧志

並行性制御法におけるROS TFの高品質化

慶應義塾大学
環境情報学部
2022年1月
荻原　湧志

並行性制御法における ROS TF の高品質化
Make ROS TF high quality in concurrency control method

学籍番号：71970013 / t19501yo
氏名：荻原　湧志
Yushi Ogiwara

Robot Operating System(ROS) はロボットソフトウェア用のミドルウェアソフトプラットフォームであり、近年多くの研究用ロボットで用いられている。TF ライブラリは ROS で頻繁に使用されるパッケージであり、ロボットシステム内の座標変換を追跡し、データを変換する標準的な方法を提供するために設計されたものである。ROS の開発初期には複数の座標変換の管理が開発者共通の悩みの種であると認識されていた。このタスクは複雑なために、開発者がデータに不適切な変換を適用した場合にバグが発生しやすい場所となっていた。また、この問題は異なる座標系同士の変換に関する情報が分散していることが多いことが課題となっていた。そこで、TF ライブラリは各座標系間の変換を有向森構造として管理し、効率的な座標変換情報の登録、座標変換の計算を可能にした。しかしながら、この有向森構造には非効率な並行性制御によりアクセスするスレッドが増えるに従ってパフォーマンスが低下する問題、及び座標変換の計算時に最新のデータを参照することができないという問題があることがわかった。そこで、我々はデータベースのトランザクション技術における細粒度ロッキング法、及び並行性制御のアルゴリズムの一種である 2PL を応用することにより、この問題を解決した。提案手法では既存手法と比べ最大 455 倍のスループットを出すことを示した。

研究指導教員：川島 英之

目 次

図 目 次

第1章　はじめに

1.1　研究背景

ロボットを使って作業を行う場合、ロボット自身がどこにいるのか、ロボットにはどこにどんなセンサーがついており、また周りの環境のどこにどんなものがあるかをシステムが把握することが重要である。例えば、図1.1 のように部屋の中にロボットと、ロボットから観測できる二つの物体があるケースを考える。図中にてロボットは円形、物体は星形で表現され、ロボットが向いている方向は円の中心から円の弧へつながる直線の方向で表される。途中で交わる二つの矢印は各座標系の位置と原点、姿勢を表す。ここでは、地図座標系、ロボットの座標系、二つの物体それぞれの座標系が示されている。

システムはロボットに搭載されたセンサーからのデータを元に各座標系間の位置関係を随時更新する。この位置関係は並行移動成分と回転成分で表現できる。例えば、自己位置推定プログラムは LiDAR から点群データが送られてくるたびにそれを地図データと比較して自己位置を計算し、ロボットが地図座標系にてどの座標に位置するか、ロボットがどの方向を向いているかといった、地図座標系からロボット座標系への位置関係を更新する。物体認識プログラムはカメラからの画像データが送られてくるたびに画像中の物体の位置を計算し、ロボット座標系から物体座標系への位置関係を更新する。

このように、各座標系間の位置関係の更新にはそれぞれ異なるセンサー、プログラムが使われる。各センサーの計測周期、及び各プログラムの制御周期は異なるため、各座標系間の位置関係の更新頻度も異なるものとなる。図1.2では、地図座標系からロボット座標系への位置関係データと、ロボット座標系から物体座標系への位置関係データがそれぞれ異なるタイミングで登録されていることを示している。

ここで地図中での物体の位置を把握するために、地図座標系から物体座標系への位置関係を取得する方法について考える。地図座標系から物体座標系への位置関係は地図座標系からロボット座標系への変換とロボット座標系から物体座標系への変換を掛け合わせれば計算ができるが、図1.2のように各変換データは異なるタイミングで来るため、最新の変換データを取得するプログラムは複雑なものとなる。A の時刻で地図座標系から物体座標系への変換データを計算しようとするとロボット座標系から物体座標系への最新の変換データを取得できるが、地図座標系からロボット座標系への変換データはまだ取得できない。このため、最新の変換データ θ を取得する、もしくは過去のデータを元にデータの補外をする必要がある。B の時刻で地図座標系から物体座標系への変換データを計算しようとすると地図座標系からロボット座標系への最新の変換データを取得できるが、ロボット座標系から物体座標系への変換データはその時間には提供されていない。このため、α と β のデータから線形補間を行う、もしくは最新の変換データ β を取得する必要がある。また、地図座標系からロボット座標系への位置関係、ロボット座標系から物体座標系への位置関係はそれぞれ別のプログラムで計算されているため、座標系同士の位置関係に関する情報は分散した状態となっている。

このように、ROS の開発初期には複数の座標変換の管理が開発者共通の悩みの種であると認識されていた。このタスクは複雑なために、開発者がデータに不適切な変換を適用し

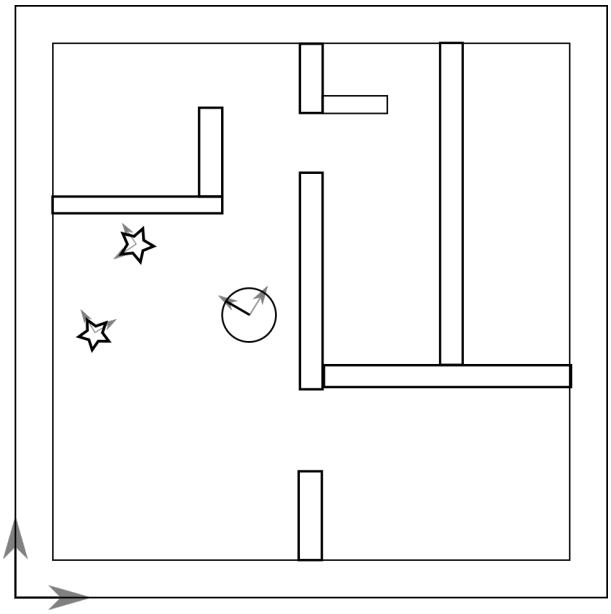

図 1.1: 部屋の中のロボット

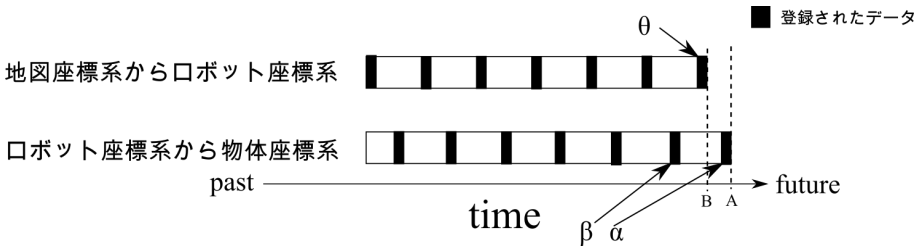

図 1.2: 位置関係の登録のタイムライン

た場合にバグが発生しやすい場所となっていた。また、この問題は異なる座標系同士の変換に関する情報が分散していることが多いことが課題となっていた。

そこで、TF ライブラリは各座標系間の変換を有向森構造として一元管理し、効率的な座標系間の変換情報の登録、座標系間の変換の計算を可能にした。まず、図1.1を表す木構造は図1.3で表現できる。木構造のノードが各座標系を表し、木構造のエッジは子ノードから親ノードへの変換データが存在することを表す。

各ノードは TF ではフレームと呼ばれ、ノード中の文字列は各座標系に対応するフレーム名が書かれている。図1.3では地図座標系のフレーム名は map、ロボット座標系のフレーム名は robot、物体 1 の座標系のフレーム名は object1 となる。子ノードから親ノードへ張られた有向エッジは子ノードから親ノードへポインタが貼られていることを表し、子ノードから親ノードを辿ることができる。このため、map から object1 への座標変換を計算するには object1 から map への座標変換の計算をし、その逆変換を取る必要がある。

子ノードから親ノードへの位置関係情報は子ノード自身が保持する。

先程説明したように、各フレーム間の座標変換情報はそれぞれ異なるタイミングで登録される。これに対処するため、TF では各フレーム間の座標変換情報を過去一定期間保存する。図1.3において各フレーム間の座標変換情報が登録されたタイミングを表すのが図1.4である。横軸は時間軸を表し、左側が過去、右側が最新の時刻を表す。黒色のセルはデータがその時刻にデータが登録されたことを表す。時刻 A では robot から map への座標変換の

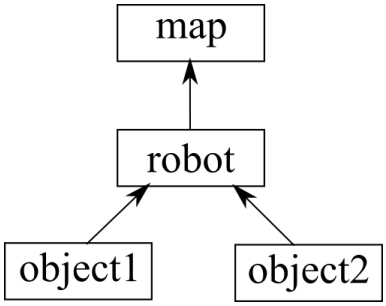

図 1.3: 図1.1に対応する木構造

情報が得られるが、object1 から robot への座標変換の情報は時刻 A には存在しない。そこで、TF では前後のデータから線形補間を行うことにより該当する時刻の座標変換データを計算する。つまり、TF は該当する時刻の座標変換データが保存されている、もしくは前後の値を元に線形補間ができる場合にはその時刻の座標変換データを提供できる、とみなす。灰色の領域は線形補間により座標変換データが提供可能な時間領域を表す。

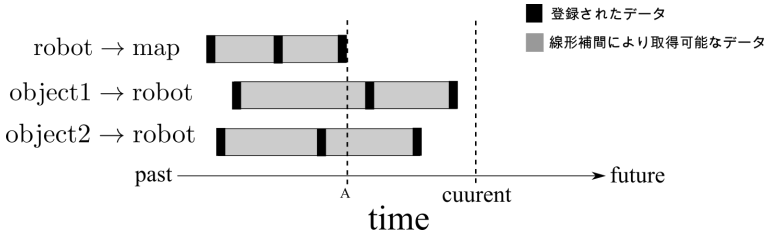

図 1.4: 図1.3における位置関係登録のタイムライン

図1.3における位置関係登録のタイムラインが図1.4のようになっているとき、TF では object1 から map への最新の位置関係は次のように計算する。

まず、object1 から map へのパスを確認する。ここでは object1 から map へのパスは object1→robot, robot→map であることがわかる。

次に、どのパスにおいてもなるべく最新の座標変換を提供できる時刻を確認する。図1.4を確認すると、object1→robot、robot→map において最新の座標変換情報が登録された時刻が最も古いのは robot→map である。このため、時刻 A がここでは要件を満たす。

最後に、時刻 A での各パスのデータを取得し、それらを掛け合わせる。robot→map については登録されたデータを使い、object1→robot については線形補間によってデータを取得する。

1.2 研究課題

前述したように TF はロボットシステム内部の座標系間の位置関係を一元管理する機構を提供する。しかしながら、これには以下のような問題点が挙げられる。

問題 1：ジャイアント・ロック

TF の森構造には複数のスレッドがアクセスするため並行性制御が必要となるが、既存の TF では一つのスレッドが森構造にアクセスしている際は他のスレッドは森構造にアクセス

できないアルゴリズムとなっており、これは、マルチコアが常識となっている現代では大きな問題となる。

問題 2：データの鮮度

上記の説明のように、TF のフレーム間の座標変換計算インターフェースは最新のデータを使わない可能性がある。同時刻のデータを元に座標変換を行うためデータの同期性はあるが、最新の座標変換データを使わないためデータの鮮度は失われる。現在、TF ライブラリには最新の座標変換データをもとにフレーム間の座標変換計算をするインターフェイスは無い。

1.3 研究方針

前述した問題 1 については、データベースの並行性制御技術における細粒度ロッキング法を適用して解決する。細粒度ロッキング法は、並行性制御においてロックするデータの単位をなるべく小さくし、並行性を向上させる手法である。

問題 2 については、データベースのトランザクション技術における 2PL を適用し、複数の座標変換の最新のデータを atomic に取得するインターフェース、及び複数の座標変換の最新のデータを atomic に更新するインターフェースを提供する。2PL とは、複数のデータに対するロック・アンロックのタイミングを二つのフェーズに分けることにより並行性を向上させつつデータ操作の一貫性を確保する手法である。

1.4 貢献

本研究ではデータベースのトランザクション技術における再粒度ロッキング法、及び並行性制アルゴリズムの一種である 2PL を応用することにより、問題 1 および問題 2 を解決した。

1.5 構成

本論文の構成は次の通りである。第二章では関連研究について述べる。第三章では既存の TF の森構造とその問題点について述べる。第四章では提案手法である森構造への再粒度ロックの導入とデータ一貫性のためのインターフェイスの提供について述べる。第五章では提案手法の評価結果を述べる。第六章では本研究の結論を述べる。第七章では今後の課題について述べる。

第2章　関連研究

データベース分野におけるロボットの研究の例としてはGAIA platform[5] が挙げられる。GAIA platform はリレーショナルデータベースと、そのデータベースに変更が加えられた時の処理を C++で宣言的に記述できる仕組みを組み合わせることにより、イベントドリブンなフレームワークでロボットや自動運転システムを構築するものである。

TF ライブラリのようにデータを時系列的に管理するライブラリとして SSM が挙げられる。SSM では各種センサデータを共有メモリ上のリングバッファで管理することにより、時刻の同期を取れたデータを高速に取得することができる。

ROS はロボットソフトウェア用のミドルウェアソフトプラットフォームであり、近年多くの研究用ロボットで用いられている。産業用途にも利用可能にするために ROS の次世代バージョンである ROS2[6] の開発が進んでいるが、並行性制御アルゴリズムは ROS から変わっておらず、本研究のようなアプローチはない。

データベース分野における高速並行性研究におけるトランザクション処理システムとして 2PL、Silo、Cicada が挙げられる。従来のトランザクション処理システムはハードウェアのコア数が少なく、メモリが小さい中でいかに効率的に処理を行うかに重きが置かれてきたため、2PL などの悲観的ロッキング法が主に使われていた。しかしながら、近年のハードウェアはメニーコア化と大容量メモリが前提のシステムとなっており、従来手法では必ずしも性能が最大限出るとは言えない。そこで Silo[8] や Cicada[9] などの楽観的並行性制御法が提案された。これらは近年のハードウェアを前提に設計されているため、それまでのシステムと比べ非常に高い性能を実現する。

第3章　既存のTFの森構造とその問題点

3.1　構造

TF ライブラリでは図3.1 のように各座標系間の位置関係を森構造で管理し、複数の木の登録が登録できる。ノードが各座標系を表し、エッジは子ノードから親ノードへの座標変換データが存在し、また子ノードから親ノードへポインタが貼られていることを表す。このため子ノードから親ノードへ辿ることはできるが、親ノードから子ノードを辿ることはできない。各ノードはフレームと呼ばれ、ノード内の文字列は各座標系に対応するフレーム名が書かれている。

各フレーム間の座標変換情報は過去 10 秒間保存される。このため、各フレーム間の座標変換情報が登録された時刻を図3.2のようなタイムラインで表現できる。黒のセルは登録されたデータを表し、灰色のセルは線形補間により座標変換データが取得可能な時刻を表す。横軸が時間軸を表し、左側が過去、右側が最新の時刻を表す。

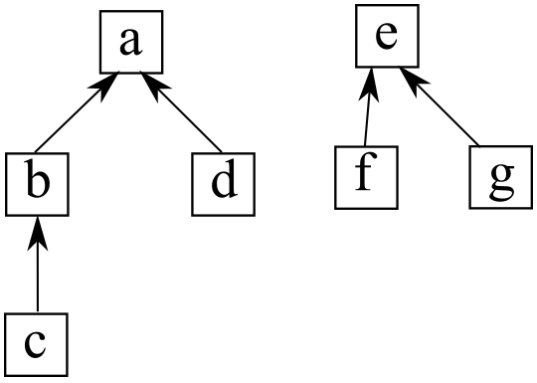

図 3.1: 複数の木構造

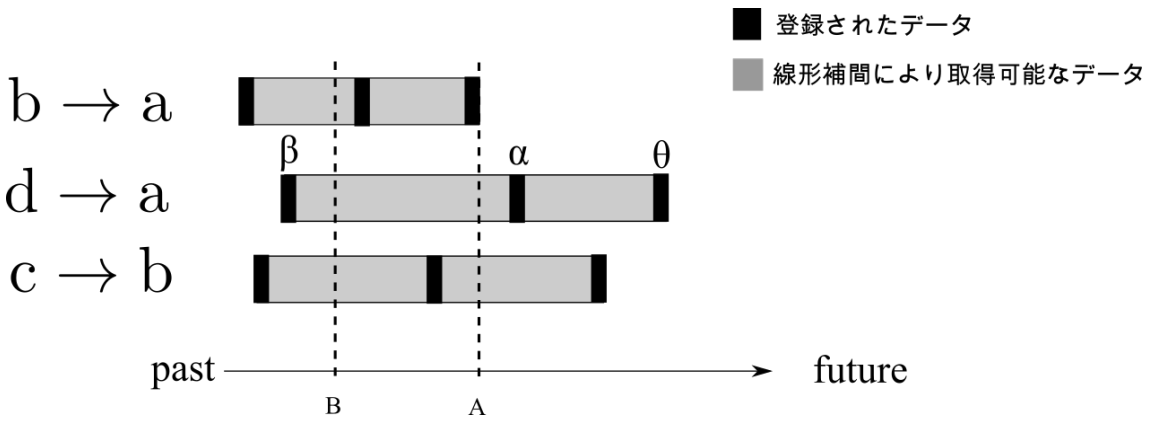

図 3.2: タイムライン

3.2 lookupTransform

二つのフレーム間の座標変換情報を取得するには lookupTransform メソッドを使う。二つのフレームが同じ木構造に属している場合にのみフレーム間の座標変換が計算できる。

登録されたフレームが図3.1、登録された座標変換情報のタイムラインが図3.2の状況において、lookupTransform メソッドを用いてフレーム c からフレーム d への座標変換を計算するアルゴリズムを説明する。

1. フレーム c から木構造のルートノードへのパスを取得する。ここではルートノードは a となり、フレーム c からフレーム a へのパスは c→b と b→a となる。
2. フレーム d から木構造のルートノードへのパスを取得する。同じようにルートノードは a となり、フレーム d からフレーム a へのパスは d→a となる。
3. 得られた三つのどのパスにおいてもなるべく最新の座標変換を提供できる時刻を確認する。ここでは時刻 A が要件を満たす。
4. 時刻 A における各パスの座標変換データを計算する。b→a については登録されたデータを利用でき、c→b と d→a については線形補間されたデータを利用できる。
5. フレーム c からフレーム a への座標変換と、フレーム a から d への座標変換を掛け合わせる。フレーム c からフレーム a への座標変換は c→b と b→a の座標変換をかけ合わせれば得られ、フレーム a から d への座標変換は d→a の逆変換から得られる。

また、lookupTransform メソッドは指定した時刻のデータを取得することもできる。時刻 B におけるフレーム c からフレーム d への座標変換は次のアルゴリズムで得られる。

1. フレーム c から木構造のルートノードへの時刻 B における座標変換を取得する。フレーム c から木構造のルートノードへのパスは c→b、b→a となり、それぞれの座標変換は線形補間によって得られる。
2. フレーム d から木構造のルートノードへの時刻 B における座標変換を取得する。フレーム d から木構造のルートノードへのパスは d→a となり、座標変換は線形補間によって得られる。
3. フレーム c からフレーム a への座標変換と、フレーム a から d への座標変換を掛け合わせる。フレーム a から d への座標変換は d→a の逆変換から得られる。

lookupTransform はアルゴリズム 1のような擬似コードで説明することもできる。

3.3 setTransform

二つのフレーム間の座標変換情報を更新するには setTransform メソッドを使う。図3.1におけるフレーム c からフレーム b のように直接の親子関係になっているフレーム間の座標変換情報を更新でき、フレーム c からフレーム a のように直接の親子関係になっていないフレーム間の座標変換情報は更新できない。フレーム c からフレーム a への座標変換を更新するにはフレーム c からフレーム b への座標変換、及びフレーム b からフレーム a への座標変換を更新すればよい。

このメソッドを呼び出すことにより新しい座標変換情報がタイムラインに追加される。

setTransform の擬似アルゴリズムをアルゴリズム3で示す。

Algorithm 1 lookupTransform

```
1: function LOOKUPTRANSFORM(target, source, time)  ▷ フレーム source からフレーム target への時刻 time での座標変換を計算する
2:     if time == 0 then  ▷ time=0 を指定すると、最新の座標変換を計算できる時刻を取得する
3:         time = getLatestCommonTime(target, source)
4:     end if
5:     source_trans = I  ▷ I は座標変換の単位元
6:     frame = source
7:     top_parent = frame
8:     while frame ≠ root do
9:         (trans, parent) = frame.getTransAndParent(time)
10:        source_trans *= trans
11:        top_parent = frame
12:        frame = parent
13:    end while
14:    frame = target
15:    target_trans = I
16:    while frame ≠ top_parent do
17:        (tarns, parent) = frame.getTransAndParent(time)
18:        target_trans *= trans
19:        frame = parent
20:    end while
       return source_trans * (target_trans)^-1
21: end function
```

Algorithm 2 getLatestCommonTime

```
1: function getLatestCommonTime(target, source) ▷ フレーム source からフレーム target への最新の座標変換を計算できる時刻を取得する
2:     frame = source
3:     common_time = TIME_MAX
4:     lct_cache = [ ]                                                    ▷ lookup tree cache
5:     while frame ≠ root do
6:         (time, parent) = frame.getLatestTimeAndParent()
7:         common_time = min(time, common_time)
8:         lct_cache.push_back((time, parent))
9:         frame = parent
10:    end while
11:    frame = target
12:    common_time = TIME_MAX
13:    while true do
14:        (time, parent) = frame.getLatestTimeAndParent()
15:        common_time = min(time, common_time)
16:        if parent in lct_cache then
17:            common_parent = parent
18:            break
19:        end if
20:        frame = parent
21:    end while
22:    for (time, parent) in lct_cache do
23:        common_time = min(common_time, time)
24:        if parent == common_parent then
25:            break
26:        end if
27:    end forreturn common_time
28: end function
```

Algorithm 3 setTransform

```
1: function setTransform(transform)                                    ▷ 座標変換 transform を登録
2:     frame = getFrame(transform.child_frame_id)
3:     frame.insertData(transform)
4: end function
```

3.4 問題点

問題 1: ジャイアント・ロック

TF ライブラリの森構造で主に使われるインターフェイスは主に lookupTransform と setTransform である。これらは複数のスレッドからアクセスされるので、並行性制御を行う必要があるが、TF ライブラリでは mutex オブジェクトを用いて森構造全体を保護している。このため、一つのスレッドが森構造にアクセスしている際は他のスレッドは森構造へのアクセスを待たされてしまう。これは、マルチコアが常識となっている現代では大きな問題となる。

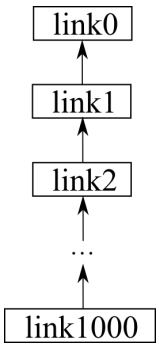

図 3.3: ヘビ型ロボットにおける森構造

例えば、ヘビ型ロボットの各関節を TF ライブラリで管理する場合について考える。関節の数が 1000 あり、各関節をフレームとして TF に登録する場合には図3.3のよう比較的巨大な森構造になる。森構造の一部のフレーム間の座標変換情報のみ更新するスレッドと、森構想の一部のフレーム間の座標変換情報のみ取得するスレッドがそれぞれ複数あった場合、それぞれのスレッドが森構造の別の部分にアクセスするにもかかわらず、一つのスレッドが森構造にアクセスするたびに森構造全体がロックされてしまいパフォーマンスに多大な影響を及ぼす。

問題 2: データの鮮度

森構造が図3.1、タイムラインが図3.2の状況において、lookupTransform を用いてフレーム c からフレーム d への最新の座標変換を計算する時には時刻 A の時点での各フレーム間の座標変換データを用いる。この時、b→a においては最新のデータを用いるが、c→b においては最新のデータと一つ前のデータから線形補間されるデータを用いている。d→a においては最新のデータ θ ではなく一つ前のデータ α とそのもう一つ前のデータ β から線形補間されるデータを用いている。このように、lookupTransform は二つのフレーム間の座標変換の計算において、フレーム間のパスの全てにおいて座標変換データを提供できる時刻についての座標変換を計算するという仕様のため、b→a のように座標変換情報の登録が遅れるとそれに足を引っ張られてしまい、最新の座標変換データが使われなくなるという問題がある。時刻の同期をとっているためデータの同期性はあるが、最新のデータが使われなくなる可能性があり、データの鮮度は失われる。現在、TF ライブラリには最新の座標変換データをもとにフレーム間の座標変換計算をするインターフェイスは無い。

第4章　提案手法

本研究では、データベースの並行性制御技術における細粒度ロッキング法及びトランザクション技術における 2PL を適用し、これらの問題を解決する。

前述した問題 1 については、データベースの並行性制御技術における細粒度ロッキング法を適用して解決する。

図4.1の森構造においてスレッド 1 が lookupTransform を用いてフレーム c からフレーム a への座標変換の計算、スレッド 2 が setTransform を用いてフレーム d からフレーム a への座標変換を更新する場合について考える。ここで、スレッド 1 はフレーム c とフレーム b のデータの読み込み、スレッド 2 はフレーム d のデータの書き込みを行うため、スレッド i のデータ x に対する読み込み操作を $r_i(x)$、スレッド i のデータ x に対する書き込み操作を $w_i(x)$ と表記すると、スレッド 1 の操作は $r_1(c)r_1(b)$、スレッド 2 の操作は $w_2(d)$ と表記できる。

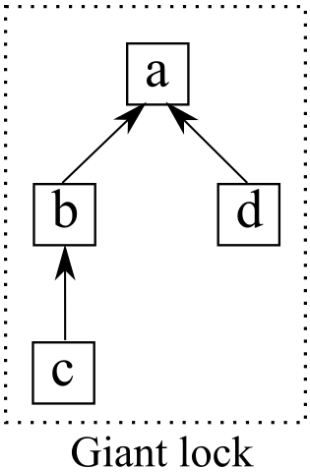

図 4.1: Giant lock

TF ライブラリでは森構造へのアクセスをする際、森構造全体をジャイアント・ロックする。これにより、図4.1の点線枠部分が保護される。図4.2はスレッド 1 の処理中にスレッド 2 の処理が開始した時のスケジュールを図示している。セルが実行中の処理を表し、セルの端の Glock と Gunlock は森構造へのジャイアントロック、アンロックを表す。スレッド 2 の処理が開始した時、スレッド 1 が森構造をジャイアントロックしているため、スレッド 2 はスレッド 1 の処理が完了し森構造のロックが外されるまで待機する必要がある。スレッド 1 がアクセスするデータとスレッド 2 がアクセスするデータは異なるため、より細かくロックする範囲を指定できる方法があればスレッド 2 がスレッド 1 の処理の完了を待つ必要がなくなる。

そこで、本研究ではデータベースの並行性制御技術における細粒度ロッキング法を適用する。細粒度ロッキング法ではアクセスするデータにのみロックをかけ、さらにロックの種類を読み込みロックと書き込みロックに分ける。複数のスレッドが同じデータにアクセスす

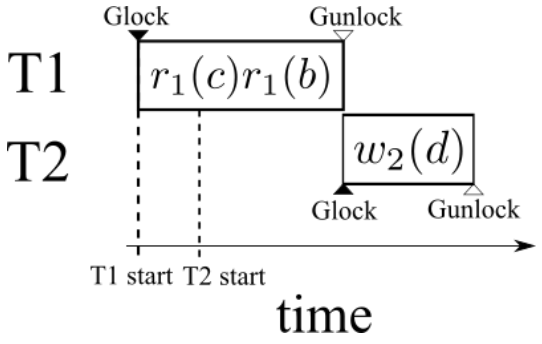

図 4.2: ジャイアントロックにおけるスケジュール

る際に発生するデータ競合を避けるために、排他制御では一つのスレッドからのみデータにアクセスできるようにするため、ロックをかける。しかしながら、複数のスレッドが同じデータにアクセスする際、データの読み込みのみ行うのであればデータ競合は発生しない。そこで、データの読み込みのみを行う時には読み込みロック、データの書き込みを行う時には書き込みロックを使い、次のようなルールを設ける。

- 読み込みを行う前に読み込みロック、書き込みを行う前に書き込みロックを行う必要がある
- ロックされていないデータには読み込みロック、及び書き込みロックをかけられる
- すでに読み込みロックされたデータにも他のスレッドが読み込みロックをかけることができる
- すでに読み込みロックされたデータには他のスレッドが書き込みロックをかけることはできない
- すでに書き込みロックされたデータには他のスレッドは読み込みロックも書き込みロックもかけることはできない

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	∘	×
$wl_j(x)$	×	×

表 4.1: lock table

このルールは、表**??**のような表形式で説明することもできる。表中ではスレッド i のデータ x に対する読み込みロック操作を $rl_i(x)$、スレッド i のデータ x に対する書き込みロック操作を $wl_i(x)$ と表記する。表は一行目がスレッド i によってロックがかけられている状態を表し、その状態に読み込みロック、または書き込みロックをスレッド $j(i \neq j)$ がかけられるかどうかを 2、3 行目で表している。∘ はすでにロックがかかっていても別のスレッドがロックをかけられることを表し、× はそうでないことを表す。例えば、2 行 2 列目はすでに $rl_i(x)$ がかかっていても $rl_j(x)$ がかけられることを表し、2 行 3 列目はすでに $wl_i(x)$ がかかっていると $rl_j(x)$ はかけられないことを表す。

細粒度ロッキングを用いた場合のスレッド 1、スレッド 2 の保護範囲は図4.3、スケジュールは図4.4で表せる。図4.4ではスレッド i がデータ x を読み込みアンロック、書き込みアンロックする時にはそれぞれ $ru_i(x), wu_i(x)$ と表記される。

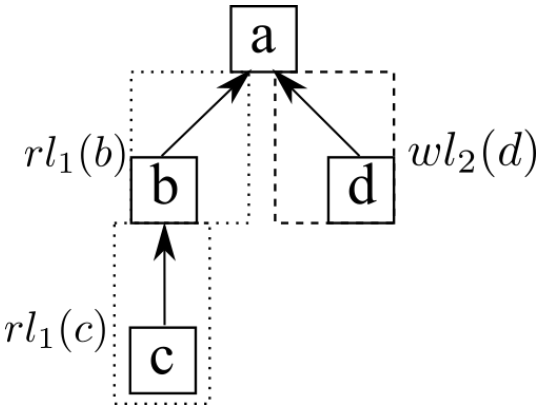

図 4.3: 細粒度ロッキング

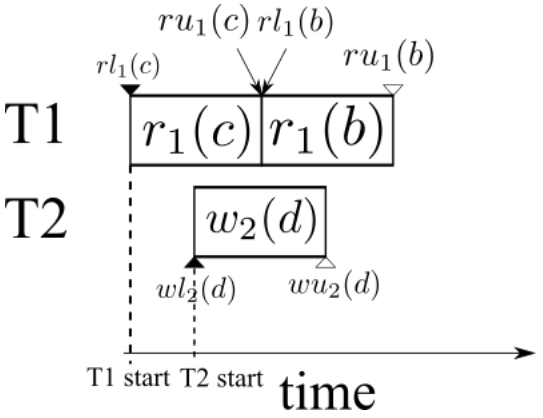

図 4.4: 細粒度ロックにおけるスケジュール

スレッド 1 の実行中にスレッド 2 の処理が開始しても、図4.3で表されるようにスレッド 1 とスレッド 2 でアクセスするデータは異なるため、スレッド 2 はスレッド 1 の処理完了を待機する必要がなくなる。このスケジュールは各操作を時系列順に表記することにより $rl_1(c)r_1(c)wl_2(d)w_2(d)ru_1(c)rl_1(b)r_1(b)wu_2(d)ru_1(b)$ と書ける。

スレッド 1 の処理の途中に、lookupTransform を用いてフレーム d のデータを読み込むスレッド 3 が開始するケースについて考える。細粒度ロッキングを用いた場合のスレッド 1 とスレッド 3 の保護範囲は図4.5で、スケジュールは図4.6で表記される。スレッド 1 にてデータ b に対して読み込みロックを取るときすでにスレッド 3 が b を読み込みロックしているが、表**??** が表すようにすでに読み込みロックがかけられていても他のスレッドが読み込みロックをかけることができる。このスケジュールは $rl_1(c)r_1(c)rl_3(b)r_3(b)ru_1(c)rl_1(b)r_1(b)ru_3(b)ru_1(b)$ と書ける。

このように、細粒度ロッキング法ではデータごとにロックをし、さらに読み込みロック・書き込みロックと区別をつけることにより並行性を上げることができる。

、細粒度ロックを実装した lookupTransform、getLatestCommonTime の擬似アルゴリズムをそれぞれアルゴリズム 4、アルゴリズム5にて示す。

前述した問題 2 については、複数の座標変換のデータを atomic に取得するインターフェース (lookupLatestTransform)、及び複数の座標変換の最新のデータを atomic に更新するインターフェース (setTransforms) を提供して解決する。setTransforms インターフェイスの必要性について説明する。

Algorithm 4 細粒度ロックを実装した lookupTransform

```
1:  function lookupTransform(target, source, time)
2:      if time == 0 then
3:          time = getLatestCommonTime(target, source)
4:      end if
5:      source_trans = I
6:      frame = source
7:      top_parent = frame
8:      while frame ≠ root do
9:          frame.rLock()
10:         (trans, parent) = frame.getLatestTransAndParent()
11:         frame.rUnlock()
12:         source_trans *= trans
13:         top_parent = frame
14:         frame = parent
15:     end while
16:     frame = target
17:     target_trans = I
18:     while frame ≠ top_parent do
19:         frame.rLock()
20:         (tarns, parent) = frame.getTransAndParent(time)
21:         frame.rUnlock()
22:         target_trans *= trans
23:         frame = parent
24:     end while
        return source_trans * (target_trans)^-1
25: end function
```

Algorithm 5 細粒度ロックを実装した getLatestCommonTime

```
1:  function getLatestCommonTime(target, source)
2:      frame = source
3:      common_time = TIME_MAX
4:      lct_cache = [ ]
5:      while frame ≠ root do
6:          frame.rLock()
7:          (time, parent) = frame.getLatestTimeAndParent()
8:          frame.rUnLock()
9:          common_time = min(time, common_time)
10:         lct_cache.push_back((time, parent))
11:         frame = parent
12:     end while
13:     frame = target
14:     common_time = TIME_MAX
15:     while true do
16:         frame.rLock()
17:         (time, parent) = frame.getLatestTimeAndParent()
18:         frame.rUnLock()
19:         common_time = min(time, common_time)
20:         if parent in lct_cache then
21:             common_parent = parent
22:             break
23:         end if
24:         frame = parent
25:     end while
26:     for (time, parent) in lct_cache do
27:         common_time = min(common_time, time)
28:         if parent == common_parent then
29:             break
30:         end if
31:     end forreturn common_time
32: end function
```

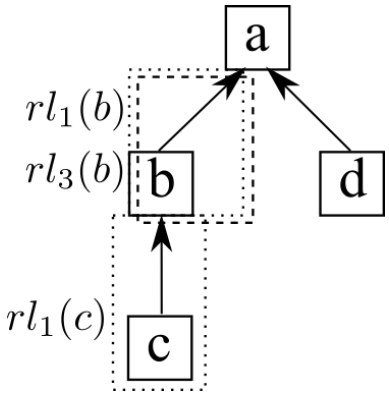

図 4.5: 二つの読み込みロック

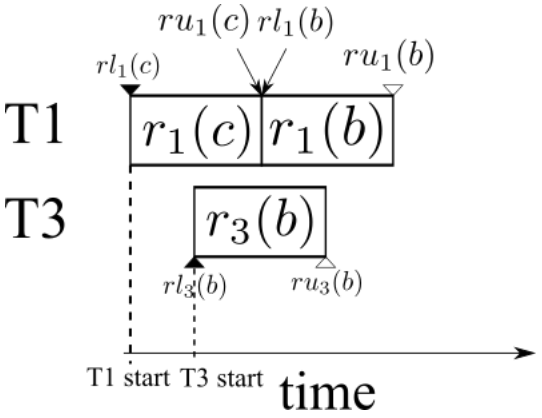

図 4.6: 二つの読み込みロックにおけるスケジュール

まず、二つのフレーム間の座標変換を計算する際に線形補間を行わずにフレーム間のパスの最新の座標変換データを使うインターフェイスとして lookupLatestTransform を導入する。これは森構造が図3.1、タイムラインが図3.2における状況でフレーム c からフレーム d への座標変換は次のように計算される。

1. フレーム c から木構造のルートノードへパスをたどりながら、各フレーム間の最新の座標変換を掛け合わせてフレームから木構造のルートノードへの座標変換を計算する。ここでは、フレーム c から木構造のルートノードへのパスは c→b、b→a となり、それぞれの座標変換は最新のものを使う。

2. 同じように、フレーム d から木構造のルートノードへパスをたどりながら、各フレーム間の最新の座標変換を掛け合わせてフレームから木構造のルートノードへの座標変換を計算する。

3. フレーム c から木構造のルートノードへの座標変換と、木構造のルートノードからフレーム d への座標変換を掛け合わせる。木構造のルートノードからフレーム d への座標変換はフレーム d から木構造のルートノードへの座標変換の逆変換から得られる。

lookupLatestTransform にて取得する座標変換データは、図4.7のように図示できる。

lookupLatestTransform を新たに提供することにより、暗黙的な線形補間をさけ、最新の座標変換データをもとにしたフレーム間の座標変換が計算できる。しかし、これには次のようなケースでは問題となる。

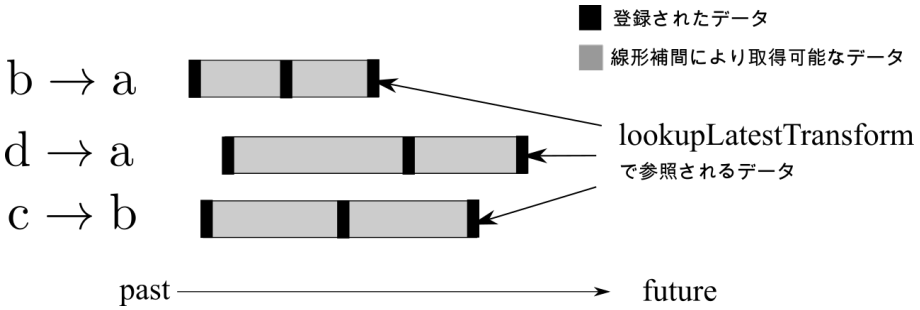

図 4.7: lookupLatestTransform で取得するデータ

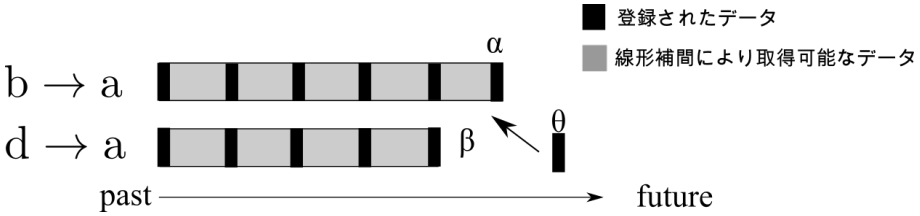

図 4.8: 同時に座標変換が登録されるケース

図4.8は森構造が図3.1の時にフレーム b からフレーム a への座標変換と、フレーム d からフレーム a への座標変換が同時刻に登録されるケースでのタイムラインを表す。b→a と a→c のデータを用いてフレーム b からフレーム c への座標変換を計算する際、ユーザーは b→a と d→a のデータについては同時刻のものを使うことを期待する。しかしながら、lookupLatestTransform を使うとユーザーの期待に反して図4.8のように θ がまだ登録されていない中間状態のタイムラインを観測し、α と β を元に座標変換してしまうことがある。これは、複数の座標変換の登録において setTransform を複数呼び出す際、 全ての座標変換が登録できていない状態で lookupLatestTransform が森構造にアクセスできることに起因する。従来の lookupTransform ではフレーム間のパスの全てにおいて座標変換データを提供できる時刻についての座標変換を計算するという仕様のため、このような問題は発生しなかった。

そこで、複数の座標変換を 2PL によって atomic に森構造に登録する setTransforms を提供し、また lookupLatestTransform も 2PL を使うように変更する。2PL[3] とは、複数のデータに対するロック・アンロックを二つのフェーズに分けることによって並行処理の結果が直列処理と同じ結果になることを保証する、データベースにおけるトランザクション技術である。このような性質は、Serializability と呼ばれる。

2PL によって並行性制御をしたときの setTransforms と lookupLatestTransform の動作について説明する。スレッド 1 が setTransforms を用いて b→a、d→a の情報を更新し、スレッド 2 が lookupLatestTransform を用いて b→a、d→a の情報を元にフレーム b からフレーム d への座標変換を計算し、スレッド 1 の処理中にスレッド 2 の処理が開始するケースについて考える。それぞれのスレッドの処理は $w_1(b)w_1(d)$、$r_2(b)r_2(d)$ と表現できる。

図4.9はスレッド 1 で $w_1(b)$ をする前に $wl_1(b)$ をした時の様子を表している。この状態でスレッド 2 の処理が始まると、$r_2(b)$ をするために $rl_2(b)$ を確保する必要があるが、まだ $wl_1(b)$ がかけられているためにロックが外されるまで待つ必要がある。

図4.10はスレッド 1 で $w_1(b)$ が完了してデータ α が登録され、$w_1(d)$ をする前に $wl_1(d)$ をした時の様子を表している。2PL では複数のロックを確保し、必ず全てのロックを取り終えてからアンロックをしていく。このため、b へのロックはまだ解放されておらずスレッド

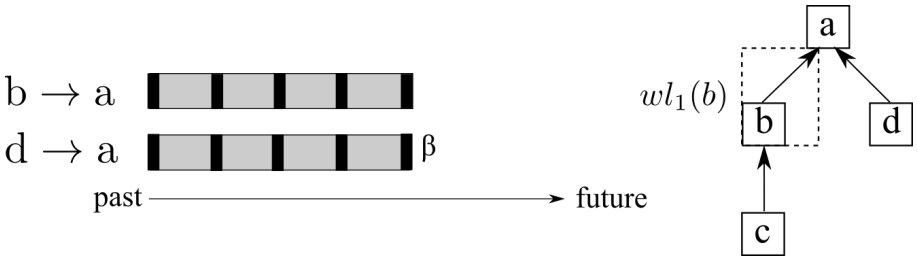

図 4.9: setTransforms と lookupLatestTransform 1

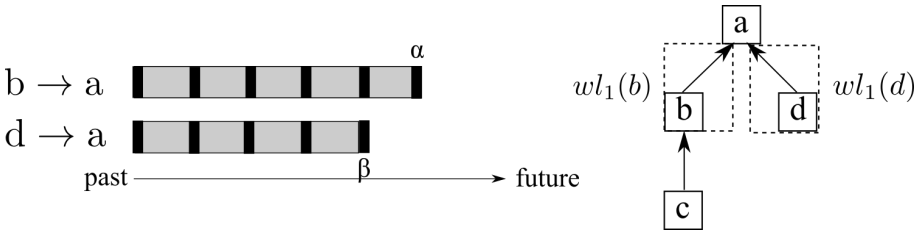

図 4.10: setTransforms と lookupLatestTransform 2

2 は待機する必要がある。

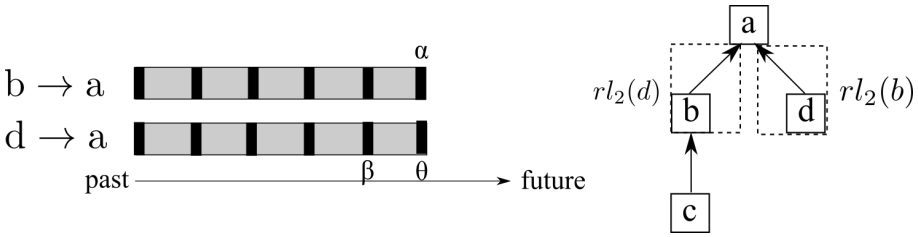

図 4.11: setTransforms と lookupLatestTransform 3

図4.11はスレッド 1 の処理が完了してデータ θ が登録され、スレッド 1 によるフレーム b,d へのロックが開放され、スレッド 2 が $r_2(b)$ と $r_2(d)$ をするために $rl_2(b)$ と $rl_2(d)$ のロックを確保した時の様子である。2PL によりスレッド 1 の処理が完了してからスレッド 2 は森構造へアクセスできるため、スレッド 2 が中間の状態を観測することはなくなる。

この一連のスケジュールは $wl_1(b)w_1(b)wl_1(d)w_1(d)wu_1(b)wu_1(d)rl_2(b)r_2(b)rl_2(d)r_2(d)ru_2(b)ru_2(d)$ と表記できる。

さて、2PL によって複数のデータに対する読み込み・書き込みが atomic に行えるようになったが、複数のデータに対してロックを取ることにより deadlock の可能性が生じる。

図4.12のような森構造において、スレッド 1 がフレーム c からフレーム d への座標変換を lookupLatestTransform を用いて計算し、スレッド 2 が setTransforms を用いて d→a 及び a→e の座標変換を更新する場合について考える。それぞれのスレッドの操作は $r_1(c)r_1(b)r_1(a)r_1(d)$、$w_2(d)w_2(a)$ と表記できる。

図4.12はスレッド 1 が a, b, c の読み込みロック、スレッド 2 が d の書き込みロックを確保した状態を表している。ここで、スレッド 1 は次に d の読み込みロックを確保したいがすでにスレッド 2 が d を書き込みロックしているためロックの解放を待機する必要がある。スレッド 2 は次に a の書き込みロックを確保したいがすでにスレッド 1 が a を読み込みロックしているために待機する必要がある。二つのスレッドがお互いのロック解放を待ち続けるた

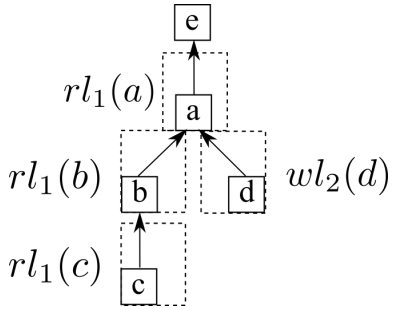

図 4.12: deadlock

め、deadlock となる。これはスレッド 2 が森構造のルートノードへ登る方向にロックをかけているのに対し、スレッド 1 は逆に一時的に森構造を下る方向にロックをかけていることに起因する。

そこで、我々は deadlock を未然に防ぐ方法として NoWait[11] を採用した。これは、書き込みロックを 2 つ以上かけようとしたときにすでにデータがロックされていたら、保持しているロックを全て解放し最初から処理をやり直す手法である。これにより、書き込みロックを 2 つ以上しているスレッドがロックの解放を待機することがなくなり、deadlock は発生しない。また、我々の手法では contention regulation として保持しているロックを全て解放して 1 ミリ秒経過してから最初から処理をやり直す。

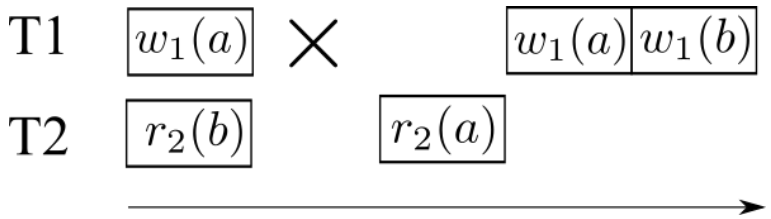

図 4.13: Dirty Read

NoWait によって処理のやり直しが発生するため、setTransforms では書き込みを行うタイミングに注意する必要がある。T1 が setTransforms を用いて $w_1(a)w_1(b)$、T2 が lookupLatestTransform を用いて $r_2(b)r_2(a)$ を実行する時、図4.13のようなスケジュールになったケースについて考える。T1 が a の書き込みを終えた後に b への書き込みロックを取ろうとするが、すでに T2 によって b への読み込みロックは取られているので、NoWait によって a へのロックを外してから処理をやり直す。T1 による処理のやり直しの前に T2 が a を読み込んでしまうと、T2 は a については T1 による更新後のデータ、b については T1 による更新前のデータを読んでしまい、並行処理の結果が直列処理と同じ結果にならなくなってしまう。この問題は、トランザクション理論においては Dirty read と呼ばれる。

Dirty read を避けるため、我々の手法では全ての書き込みロックが確保できてから座標変換の書き込みを行うようにした。これにより、書き込みが一部行われた状態を読み込み専用スレッドが観測することはなくなる。

setTransforms と lookupLatestTransform を利用すると、最新の座標変換を更新・取得できるだけではなく、無駄な座標変換の更新も減らすことができる。図4.14において、b→a はあまり座標系間の位置関係が変わらないために座標変換はあまり更新されないが、d→a の座標変換は頻繁に更新されるケースについて考える。座標変換の計算において b→a と d→a のデータを用いる場合、既存の TF では「該当する時刻の座標変換データが保存されている、

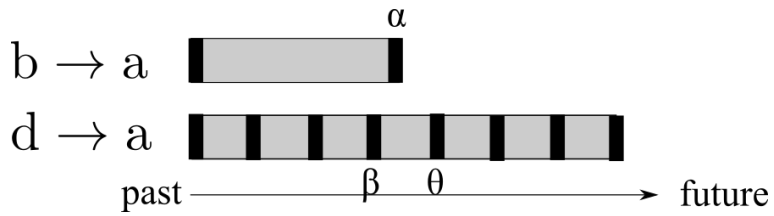

図 4.14: deadlock

もしくは前後の値を元に線形補間ができる場合にはその時刻の座標変換データを提供できると見做す」という仕様により、b→a の更新が遅いために d→a では過去の鮮度の低い β と θ から座標変換の計算をしなくてはならない。これを避けるため、既存の TF ではあまり座標系間の位置関係が変わらない b→a においても、一定周期で同じ座標変換情報を登録する必要があった。このように、既存の TF では座標変換情報が変わらないにもかかわらず一定周期で同じ座標変換情報を登録する必要があり、余計な負荷がかかっていた。しかしながら、setTransforms と lookupLatestTransform では最新の座標変換のみを見るため必要な時にのみ座標変換の更新をすればよく、このような余計な負荷がかかることは無くなる。

lookupLatestTransform、setTransforms の擬似アルゴリズムをそれぞれアルゴリズム6、7にて示す。

Algorithm 6 lookupLatestTransform

```
1:  function LOOKUPLATESTTRANSFORM(target, source)
2:      rlock_list = [ ]
3:      source_trans = I
4:      frame = source
5:      top_parent = frame
6:      while frame ≠ root do
7:          rlock_list.push_back(frame)
8:          (trans, parent) = frame.getLatestTransAndParent()
9:          source_trans *= trans
10:         top_parent = frame
11:         frame = parent
12:     end while
13:     frame = target
14:     target_trans = I
15:     while frame ≠ top_parent do
16:         rlock_list.push_back(frame)
17:         (tarns, parent) = frame.getTransAndParent(time)
18:         target_trans *= trans
19:         frame = parent
20:     end while
21:     unlock all in rlock_list
          return source_trans * (target_trans)^-1
22: end function
```

Algorithm 7 setTransforms

```
1:  function setTransforms(transforms)
2:      wlock_list = [ ]
3:      for trans in transforms do
4:          frame = getFrame(trans.child_frame_id)
5:          lock_success = frame.tryWLock()
6:          if lock_success then
7:              wlock_list.push_back(frame)
8:          else
9:              unlock all in wlock_list
10:             sleep 1ms
11:             goto 2
12:         end if
13:     end for
14:     for trans in transforms do ▷ Dirty read を避けるため、全ての wlock が確保できてから書き込み
15:         frame = getFrame(trans.child_frame_id)
16:         frame.insertData(trans)
17:     end for
18:     unlock all in wlock_list
19: end function
```

第5章　評価

TF ライブラリに細粒度を実装した lookupTransform・setTransform、及び最新の複数のデータを atomic に取得・更新できる lookupLatestTransform・setTransforms を以下の指標で評価した。

1. スループット: 一秒間に何回操作 (lookupTransform 及び setTransforms) ができたか
2. レイテンシ: 操作の応答時間
3. データの鮮度: lookupTransform においてアクセスした各座標変換データのタイムスタンプの新しさ
4. データの同期性: lookupTransform においてアクセスした各座標変換データのタイムスタンプの同一性
5. abort 率 (setTransforms のみ): setTransforms において、NoWait によって操作をやり直した回数の比率。(やり直した回数 / setTransforms を呼び出した回数) で求める。

以下、既存手法は old、細粒度ロックを実装した lookupTransform・setTransforms を snapshot、2PL を用いて複数のデータを atomic に取得・更新できる lookupLatestTransform・setTransforms を latest と呼称する。

5.1　実験環境

実験には Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz を 4 つ搭載したサーバを利用する。それぞれのコアは 32KB private L1d キャッシュ、1024KB private L2 キャッシュを持つ。単一プロセッサの 28 コアは 39MB L3 キャッシュを共有し、ハイパー・スレッディングを有効化している。トータルキャッシュサイズはおよそ 160MB である。メモリは DDR4-2666 が 48 個接続されており、一つあたりのサイズは 32GB、全体のサイズは 1.5 TB である。全ての実験において，実行時間は 60 秒という安定的な結果が得られる時間を選択している。

5.2　ワークロード

実験のワークロードは、関節数が多いヘビ型ロボットの関節情報を TF に登録することを想定し、図3.3のようなフレーム間の座標変換情報が一直線に与えられた構造に複数のスレッドからアクセスし計測を行う。TF ライブラリへのアクセスパターンとしては、主に lookupTransform のみを複数回呼び出すスレッドと setTransform のみを複数回呼び出すスレッドに二分される。このため、lookupTransform のみを複数回呼び出すスレッド (読み込み専用スレッド)、及び setTransform のみを複数回呼び出すスレッド (書き込み専用スレッド) をそれぞれ複数立ち上げ計測を行う。

実験においては以下のパラメータが存在する。

1. thread: 合計スレッド数
2. joint: フレームの数
3. read_ratio: 合計スレッド数のうち、読み込み専用スレッドの割合
4. read_len: 読み込み専用スレッドにて一回の lookupTransform 操作で読みこむフレームの数。joint 個のフレームのうちランダムに i 番目のフレームが選択され、そこから i+read_len 番目のフレームまでの座標変換が計算される。
5. write_len: 書き込み専用スレッドにて一回の操作で座標変換情報を更新するフレームの数。joint 個のフレームのうちランダムに i 番目のフレームが選択され、そこから i+read_len 番目のフレームまでの座標変換が更新される。setTransform では一度に一個のフレームしか更新できないため write_len 回 setTransform を呼び出す操作を一つの操作とする。
6. frequency: 各スレッドにて操作を呼び出す周期。操作の呼び出しが完了したのち、1 / frequency 秒待機してから再び操作を呼びだす。0 に設定すると待機なしで操作を呼び出し続ける。各スレッドが一定の周期にて操作を呼び出すというのは、ROS において一般的なワークロードである。
7. opposite_write: 本実験において、読み込み専用スレッドは図5.1のように下から上に順に読み込みロックをとる。このフラグを true にすると、上から下の方向に書き込みロックをとり、読み込みロックとは逆方向にロックをとる。false にすると下から上の方向に書き込みロックをとる。

各実験はそれぞれ YCSB-A/B/C[10] ワークロードについて行われている。YCSB-A/B/C はそれぞれ、読み込み操作と書き込み操作の割合が 50:50、95:5、100:0 のワークロードを指す。ここでは読み込み専用スレッドの数と書き込み専用スレッドの数の比でそれぞれのワークロードを再現するため、read_ratio をそれぞれ 0.5、0.95、1 に設定した。

特に記載がない場合は joint=10000、read_len=16、write_len=16、frequency=0 で実験が行われている。

5.3 YCSB-C

スループットについては図5.2のように、old に比べて snapshot は最大 455 倍、latest は最大 393 倍のスループットとなった。

レイテンシについては図5.3、図5.4のように、どの手法においても線形比例しているが old に比べ snapshot、latest は非常に小さいレイテンシとなった。

スループット、レイテンシのどちらにおいても snapshot と latest が優れているのは latest は 2PL によって一度に複数のデータにロックをかけるのに対し、snapshot では最大一個のデータしかロックしか取らないため、より並行性が高くなると考えられる。

スレッドが 112 の部分にて、snapshot 及び latest のグラフに傾きが生じているのは、ハイパー・スレッディングによるものだと考えられる。

書き込みが発生しないため、YCSB-C におけるデータの鮮度、データの同期性、abort 率については説明しない。

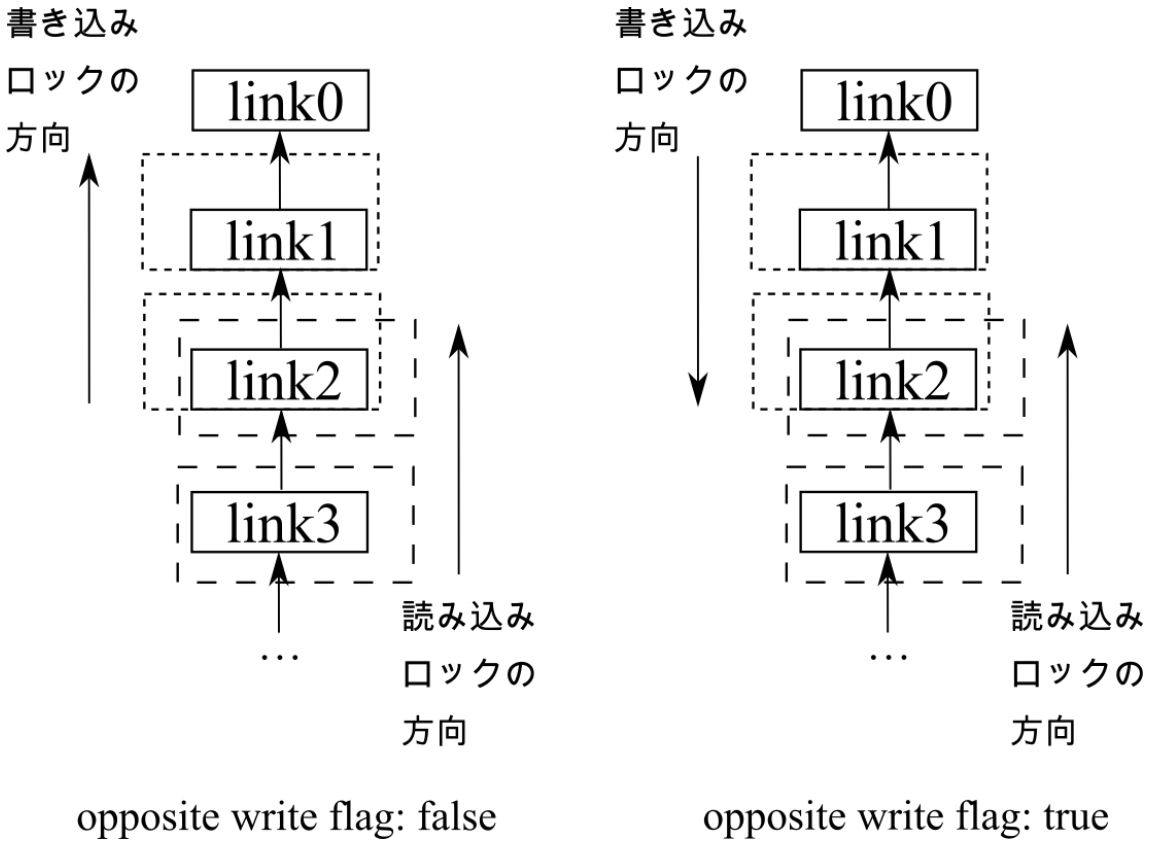

図 5.1: opposite_write フラグ

5.4 YCSB-A

YCSB-C とは異なり、YCSB-A では書き込み操作が発生するため、opposite_write フラグを有効にした場合とそうでない場合に差が生じた。

opposite_write フラグを有効にした場合ではスループットについては図5.5のように old に比べて snapshot は最大 91.3 倍、latest は最大 203 倍のスループットとなり、latest のスループットが snapshot よりも高くなった。

これに対し、opposite_write フラグを無効にした場合ではスループットについては図5.6のように old に比べて snapshot は最大 92 倍、latest は最大 94 倍のスループットとなった。

どちらの場合においても snapshot はほとんど変わらないが、latest のスループットには大きな変化が現れた。

なぜこのようになったかを調べるために、まずは abort 率について図5.7、5.8に表示した。どちらもスレッド数に比例して abort 率が上がっているが、opposite_write が有効になっている方が全体的に abort 率が高く、書き込みロックを読み込みロックの逆方向に確保していくと abort しやすいことがわかる。

これは、読み込みロックと同じ方向に書き込みロックをかけた場合には、一番最初の要素に書き込みロックができればその上の要素に abort なしで書き込みロックを取れる可能性が高いのに対し、逆方向に書き込みロックをかけた場合にはいくつかの要素の書き込みロックが取れていても、他の読み込み専用スレッドと衝突し、全ての書き込みロックを外す必要があるため、このような abort 率の差が生じたと考えられる。

読み込みロックと同じ方向に書き込みロックをかけた場合には読み込み専用スレッドは

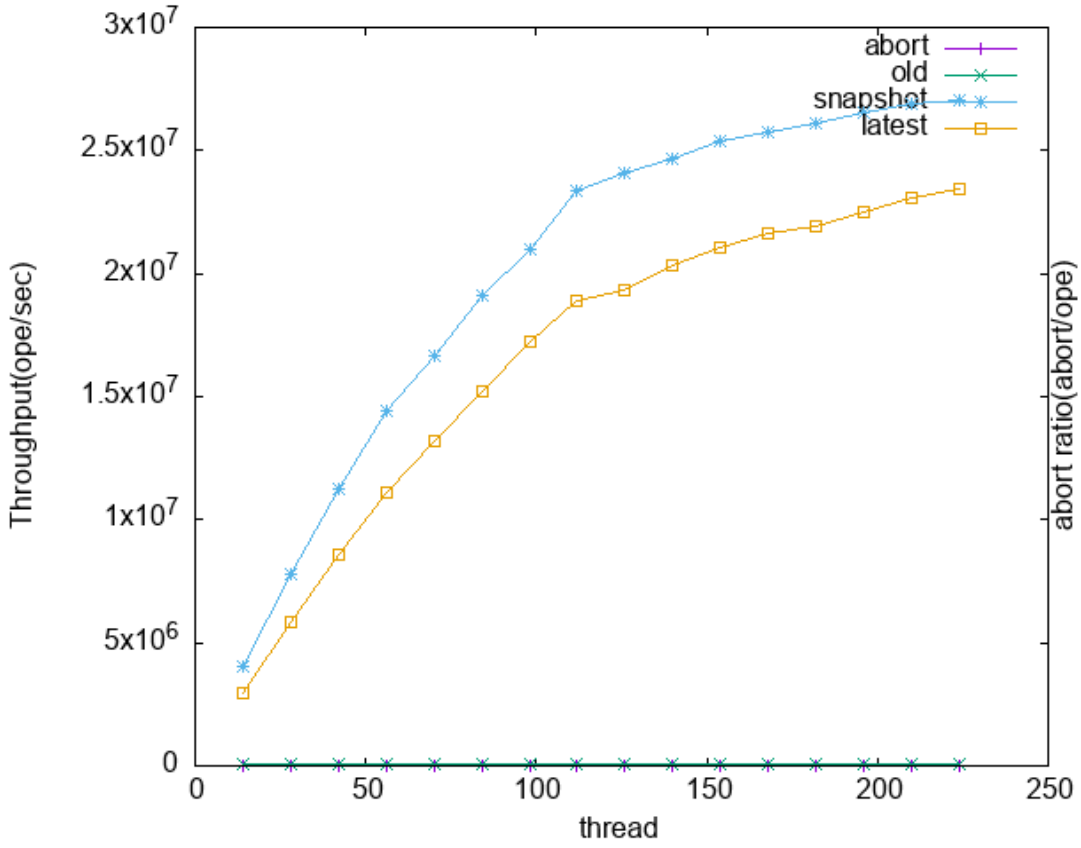

図 5.2: YCSB-C におけるスレッド数とスループットの関係

書き込み専用スレッドの完了を待機する必要があるが、逆方向に書き込みロックをかけた場合には書き込み専用スレッドが abort するために待機の必要はなくなると考えられる。

次に、読み込み・書き込み専用スレッドそれぞれのレイテンシについて図5.9〜5.16に表示した。

opposite_write フラグの有効・無効、読み込み・書き込みに関わらず、どのケースにおいても snapshot、latest の方がレイテンシが低いことがわかる。また、opposite_write フラグの有効・無効に関わらず、大きな変化がないことがわかる。

図5.10のように、opposite_write フラグが有効な時に読み込みスレッドのレイテンシが latest が snapshot より早いのは、snapshot の lookupTransform において二つのフレーム間のパス上の全てのフレームにおいて座標変換を取得できる時刻を検索してから座標変換を計算するために、森構造を 2 度読み込む必要があるからだと考えられる。

図5.12のように、opposite_write フラグが有効な時に書き込みスレッドのレイテンシが snapshot が latest より早いのは、latest の setTransforms では一度に複数の要素のロック、書き込みを行い、また NoWait によって abort し処理をやり直す可能性があるからだと考えられる。

図5.14のように、opposite_write フラグが無効な時に読み込みスレッドのレイテンシが latest が snapshot とあまり変わらないのは、latest にて書き込みスレッドによる abort が減り、代わりに読み込みスレッドの待機時間が増えたからだと考えられる。これは opposite_write フラグが有効な時とは対照的である。

図5.16のように、opposite_write フラグが無効な時に書き込みスレッドのレイテンシが latest が snapshot とあまり変わらないのは、latest の abort が減ったからだと考えられる。これも

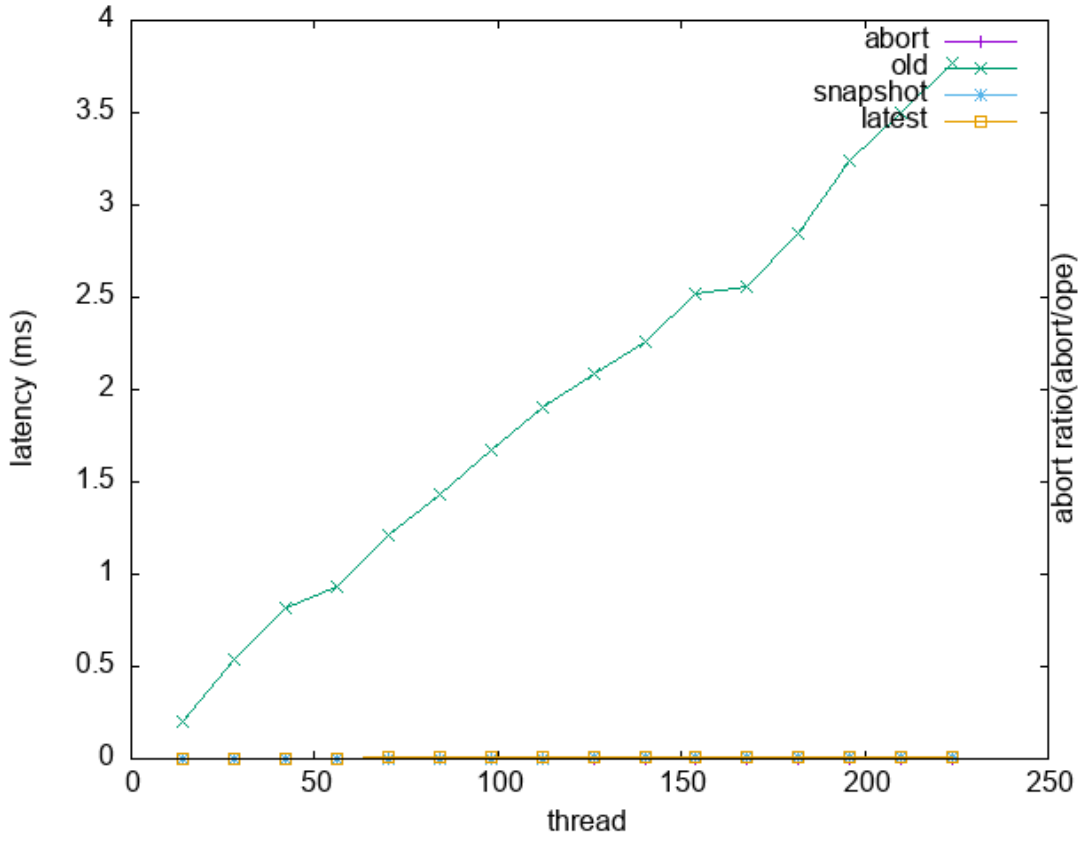

図 5.3: YCSB-C におけるスレッド数とレイテンシの関係

opposite_write フラグが有効な時とは対照的である。

latest の読み込みスレッドのレイテンシは opposite_write フラグが有効な時には最大 0.007ms 程度、opposite_write フラグが無効な時には最大 0.02ms 程度となる。これに対し、書き込みスレッドのレイテンシは opposite_write フラグが有効な時には最大 0.8ms 程度、opposite_write フラグが無効な時には最大 0.3ms 程度となる。

書き込みのレイテンシと読み込みのレイテンシの倍率は opposite_write フラグが有効な時には約 100 倍、無効な時には約 10 倍となり、書き込みの方が読み込みよりも時間がかかることがわかる。また、opposite_write フラグを有効にすると読み込みのレイテンシは減るが書き込みのレイテンシが増えることがわかる。このことから、opposite_write フラグの有効・無効化の選択は読み込みのレイテンシを下げるか、それとも書き込みのレイテンシを下げるかのトレードオフとなると考えられる。

読み込み・書き込み専用スレッドそれぞれのスループットについて図5.17〜5.20に表示した。

最後にスループットについて

write のスループットが上がる代わりに read のスループットが下がると考えられる。

なぜこのようになったか調べるために、まずは読み込み・書き込み専用スレッドそれぞれのスループットを

最後にレイテンシについて

これにより、write は read に比べて時間がかかるため、書き込み順序をぎゃくにすると abort しやすくなるけど逆に read が増えて全体のスループットは上がる snapshot ではたくさ

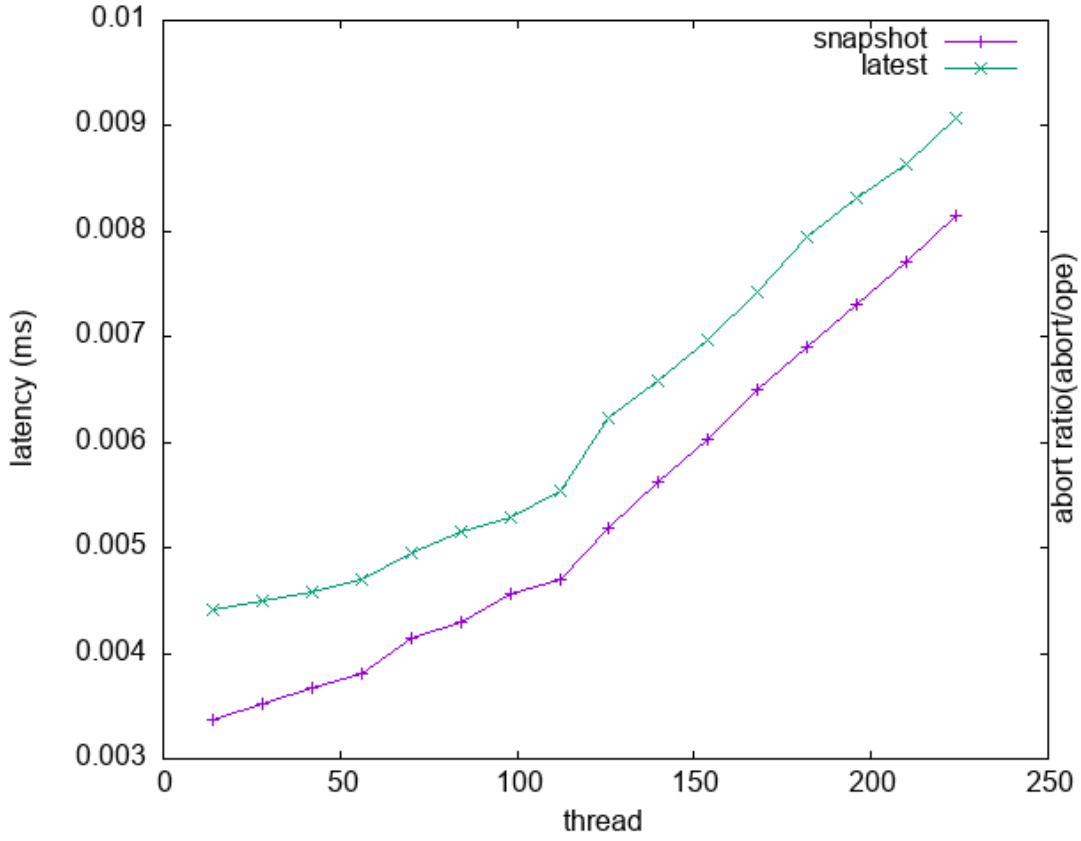

図 5.4: YCSB-C におけるスレッド数とレイテンシの関係 snapshot と latest のみ

ん呼び出ししなきゃいけない。だからレイテンシは低いけどスループットは下がる。
と考えられる。
データの鮮度はこう
データの同期性についてはこうなった。

5.5 スループット

YCSB-A/B/C ワークロードを用いて、old、snapshot、latest のそれぞれにおいてスレッド数とスループットの関係について分析する。

YCSB-A においては latest は old に比べ最大 218 倍のスループットとなり、210 スレッドまでスケールした。snapshot は old に比べ最大 49 倍のスループットとなり、96 スレッドまでスケールした。

YCSB-B においては latest は old に比べ最大 339 倍のスループットとなり、224 スレッドまでスケールした。snapshot は old に比べ最大 101 倍のスループットとなり、98 スレッドまでスケールした。

latest が snapshot より高いスループットを出しているのは、snapshot では lookupTransform において二つのフレーム間のパス上の全てのフレームにおいて座標変換を取得できる時刻を検索してから座標変換を計算するために、森構造を 2 度読み込む必要があるからだと考えられる。

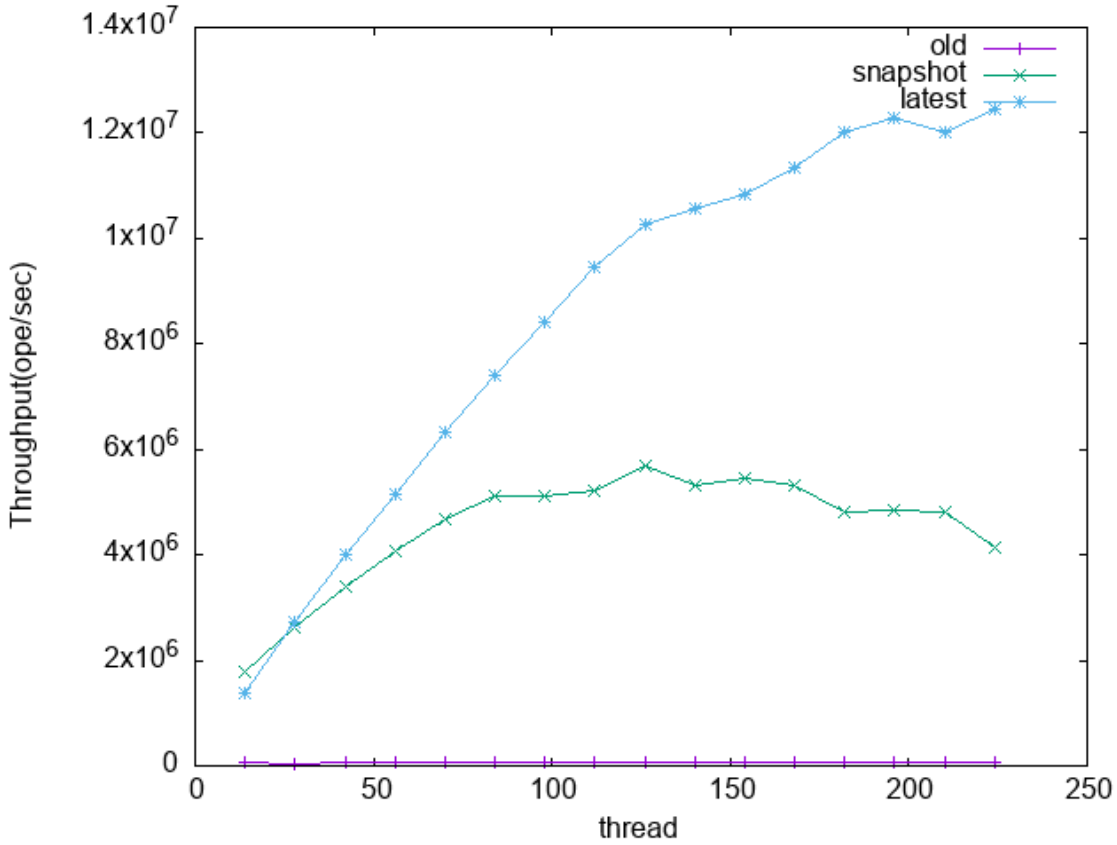

図 5.5: opposite_write フラグを有効にした場合

どのワークロードにおいても snapshot、latest は細粒度ロックの導入によって並行性を高め、old よりも高いスループットを記録した。これにより、細粒度ロックの導入はスループットの向上に大幅に貢献したと考えられる。

5.6 レイテンシ

YCSB-A/B/C ワークロードを用いて、old、snapshot、latest のそれぞれにおいてレイテンシとスループットの関係について分析する。

old ではレイテンシとスレッド数が線形比例するのに対し、提案手法ではほとんど影響がない。これは、old においてはジャイアントロックによって逐次的に操作を実行する必要があったが、snapshot と latest においては細粒度ロックにより並行に操作を実行することが可能だからである。

どのワークロードにおいても snapshot、latest は細粒度ロックの導入によって並行性を高め、old よりも低いレイテンシを記録した。これにより、細粒度ロックの導入はレイテンシの低下に大幅に貢献したと考えられる。

図5.21は、スレッド数 200 の状態で frequency を 100 から 100000 まで変化させた時のレイテンシを表している。制御周期を増やすと同時に走る操作も増えるため、このようになるのは自明である。

old においてレイテンシが問題となるワークロードを示したのが図5.22である。これは、thread=200、joint=1000000、read_ratio=0.5、read_len=10000、write_len=10000 におけるレイ

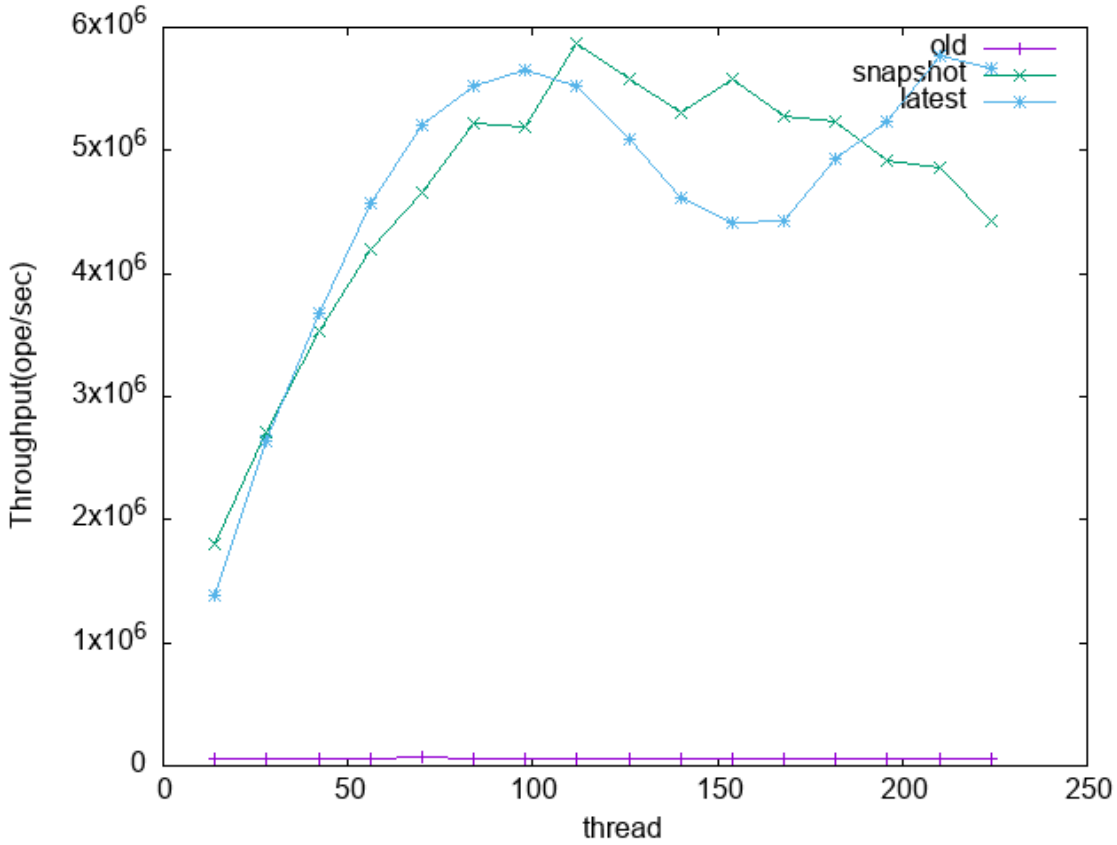

図 5.6: opposite_write フラグを無効にした場合

テンシを表している。

frequency は 100 なのでかなり現実的 snapshot と latest では 20ms、12ms なのに対し、old では 608ms となり現実的なワークロードでは問題となる。ここは joint 数を増やした時のレイテンシの変化を記録する？

この説明は本当にいるのか？ヘビ型ロボットや、センサで認識した物体を TF に登録するようなワークロードでは問題となる。その場合には、どんなオブジェクトがどこにあるかを統一的に扱う枠組みが必要だろう。例えば今はでもその情報は一つのマップには集約されていない。別々のプロセスが別々のデータ型で publish し、内部の情報は open ではない。例えば障害物は move_base による costmap、信号はカメラによる YOLO、人や車の検知は

5.7 データの鮮度

ycsb-c においてはデータの更新がされないため、どの手法でも同じような delay となる。

YCSB-A/B のどちらにおいても、snapshot はスレッド数の増加に伴いデータの鮮度が低下しているが、latest は鮮度がスレッド数が増えるにつれ向上している。

5.8 データの同一性

スレッド数が増えるに従って相対的に書き込み専用スレッドも増えるため、偏差も減ると考えられる。YCSB-B の方が偏差が大きいのは YCSB-A に比べて書き込み専用スレッドの

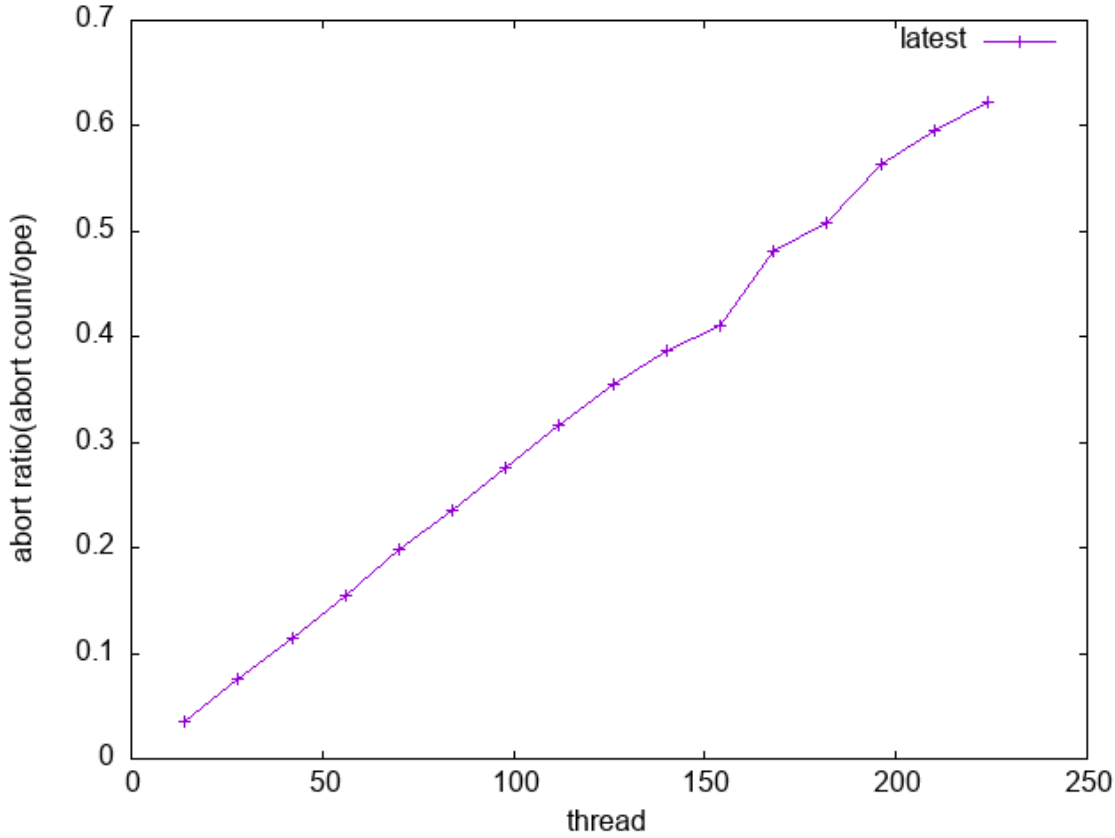

図 5.7: opposite_write フラグを有効にした場合

相対的な割合が常に小さいから。

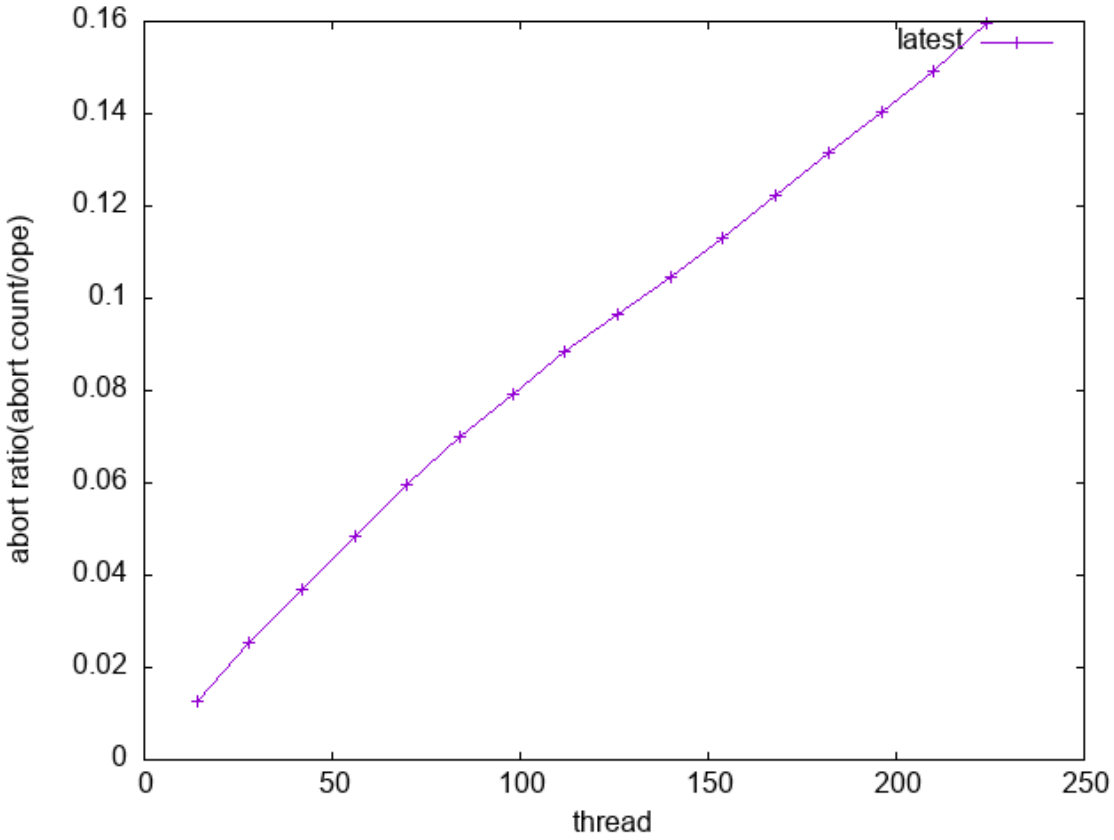

図 5.8: opposite_write を無効にした場合

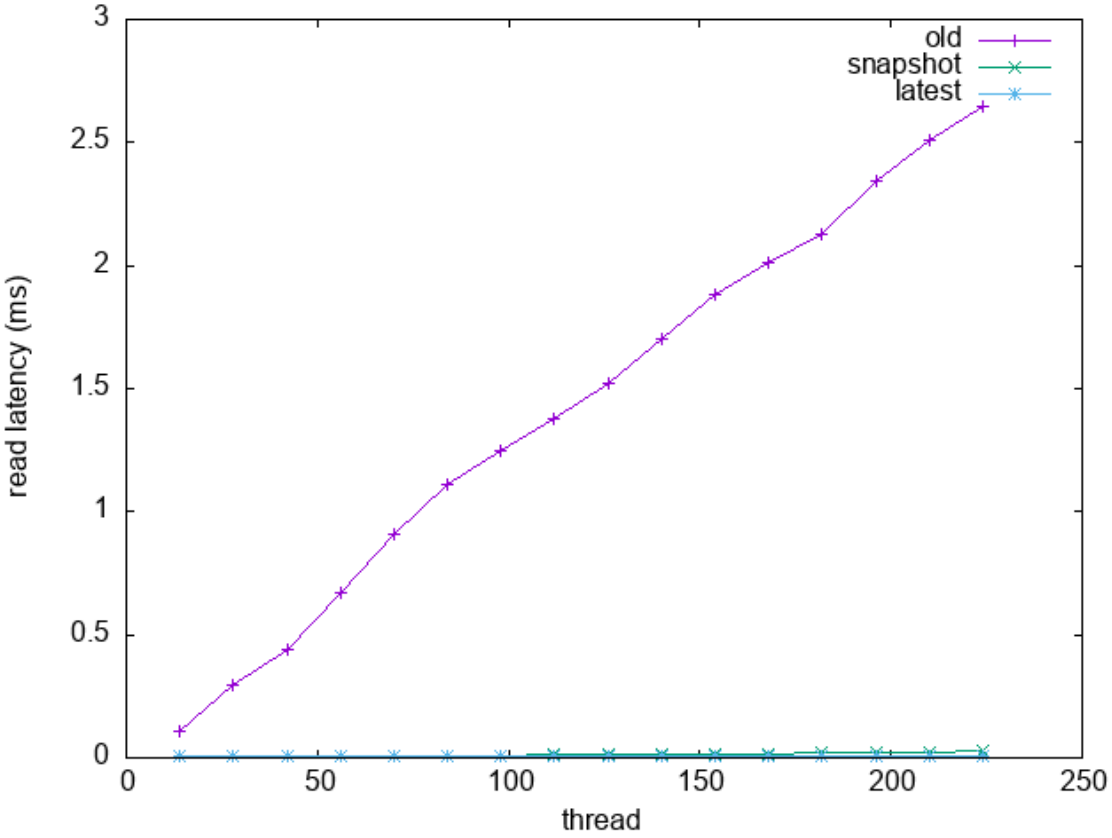

図 5.9: opposite_write フラグを有効にした場合

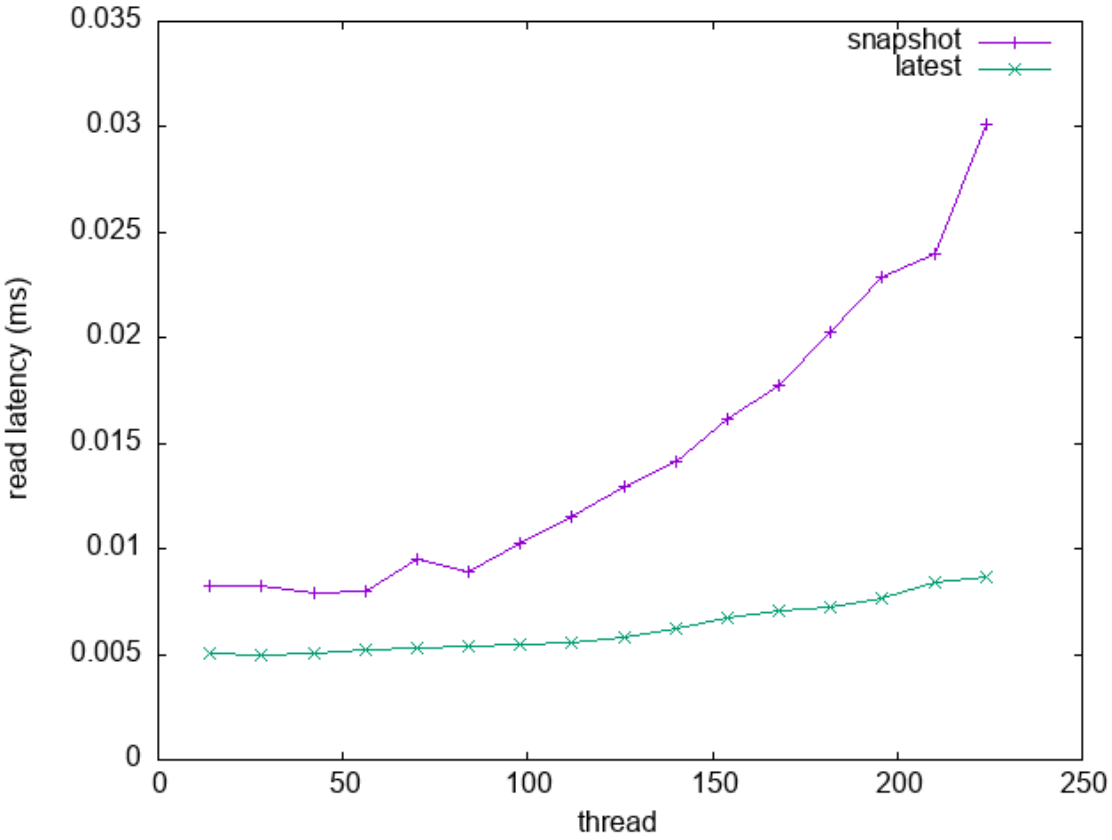

図 5.10: opposite_write フラグを有効にした場合 snapshot と latest のみ

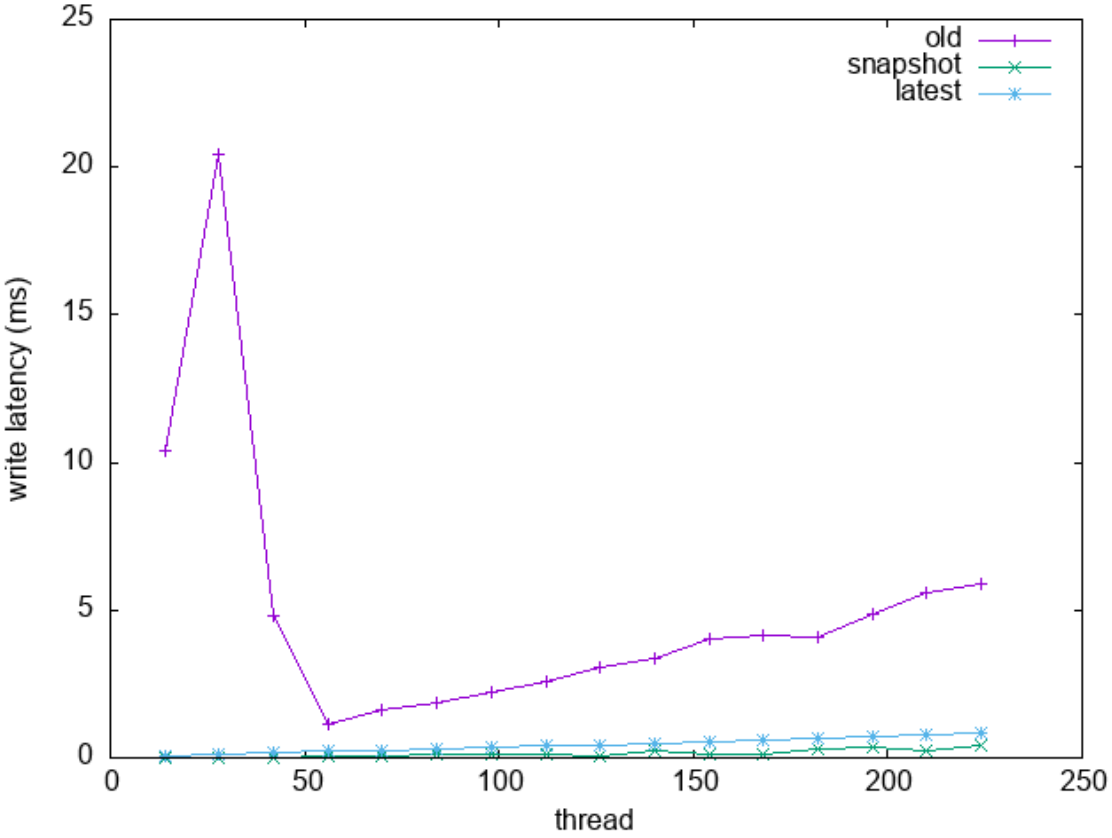

図 5.11: opposite_write フラグを有効にした場合

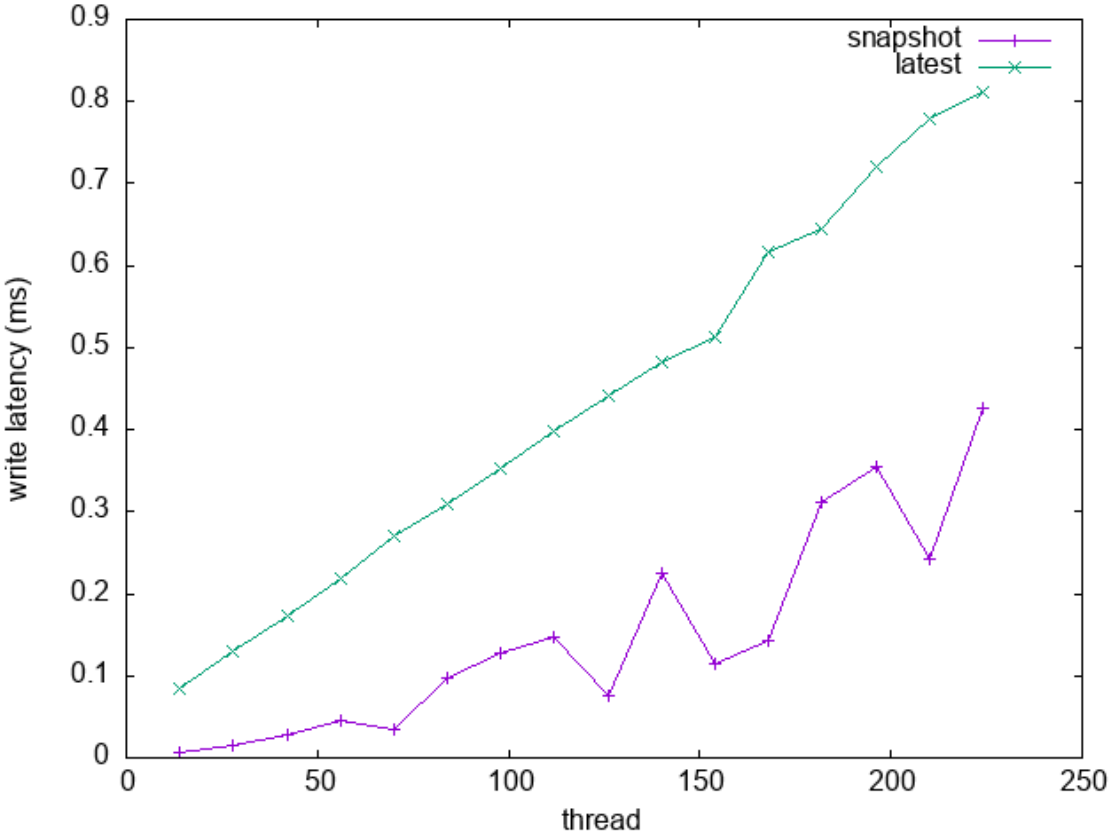

図 5.12: opposite_write フラグを有効にした場合 snapshot と latest のみ

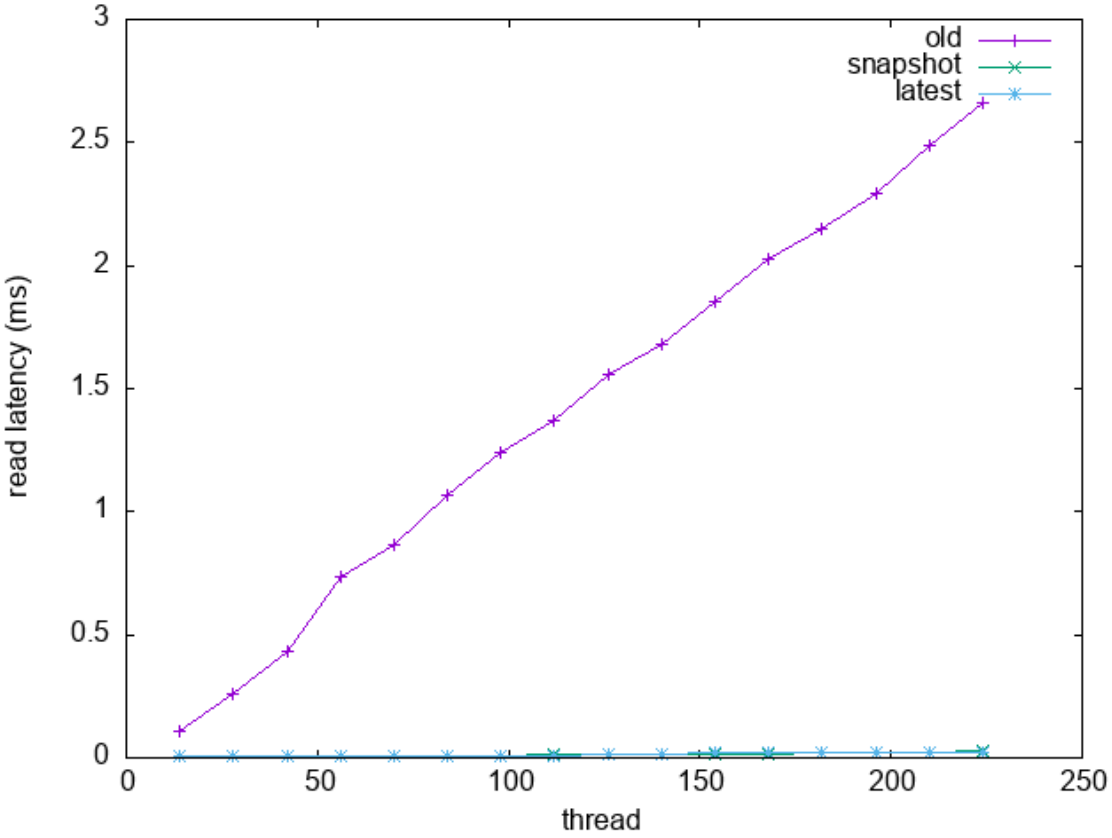

図 5.13: opposite_write フラグを無効にした場合

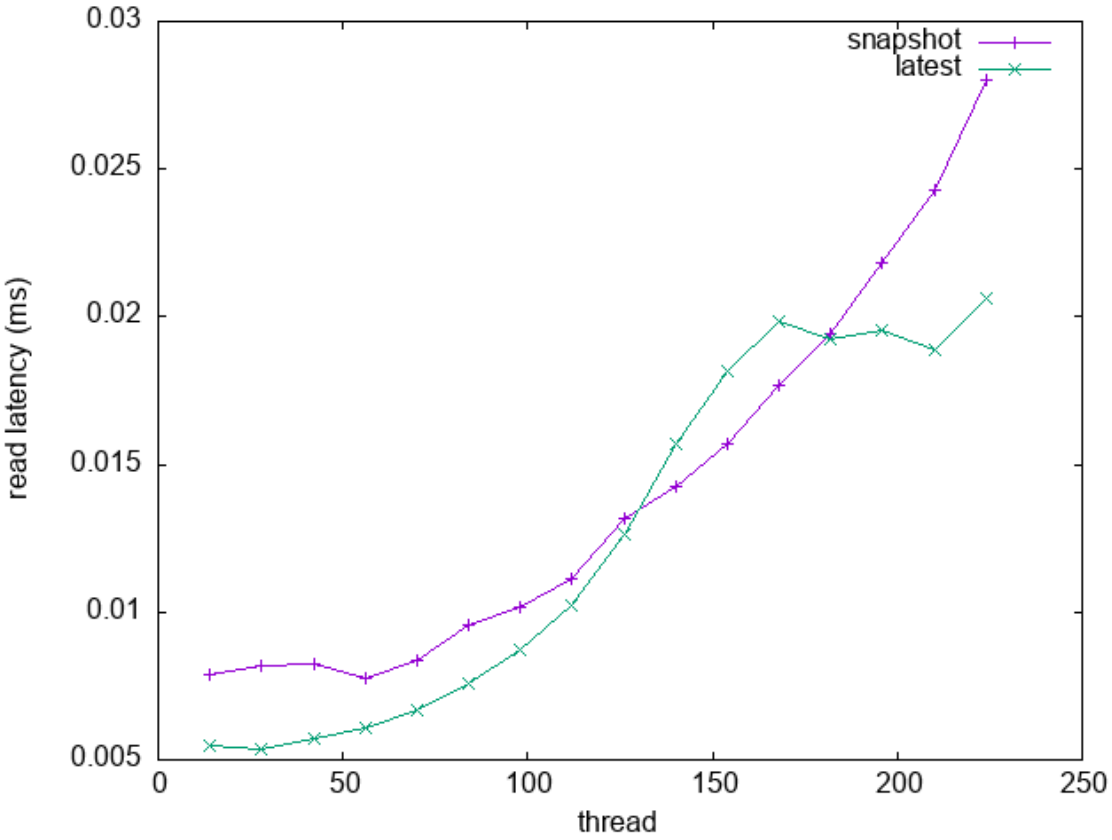

図 5.14: opposite_write フラグを無効にした場合 snapshot と latest のみ

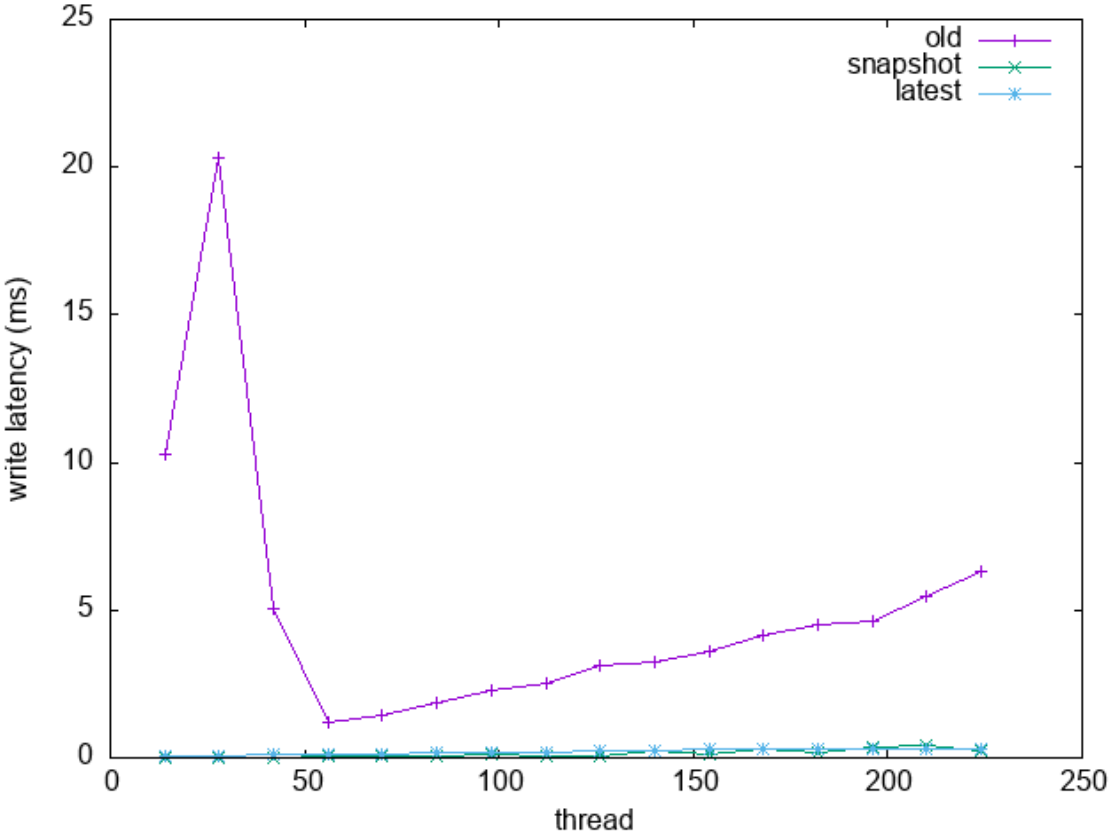

図 5.15: opposite_write フラグを無効にした場合

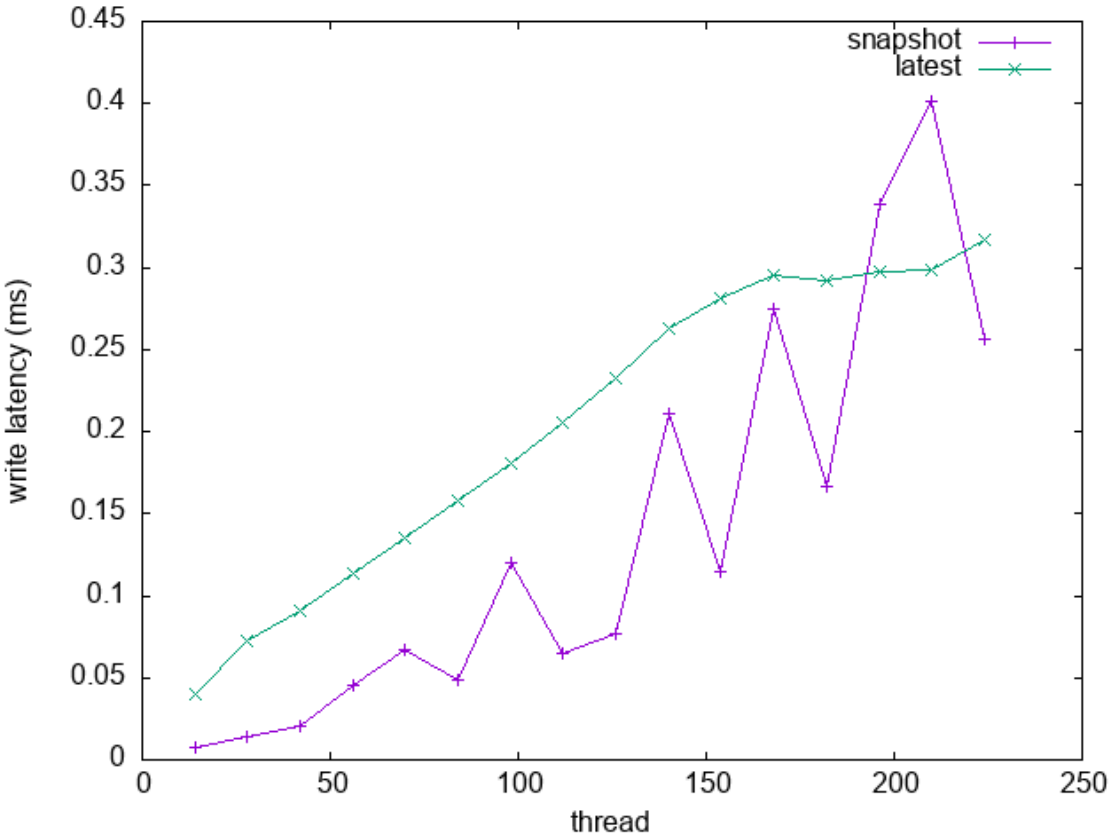

図 5.16: opposite_write フラグを無効にした場合 snapshot と latest のみ

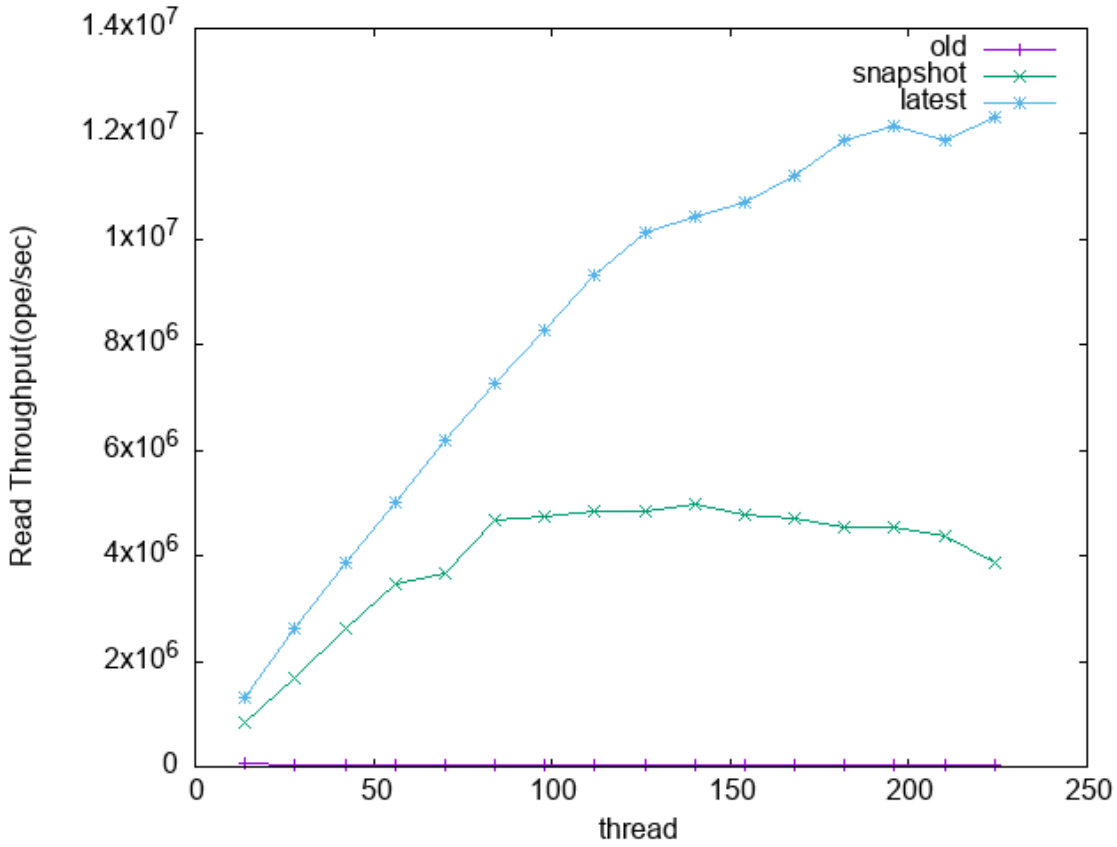

図 5.17: opposite_write フラグを有効にした場合

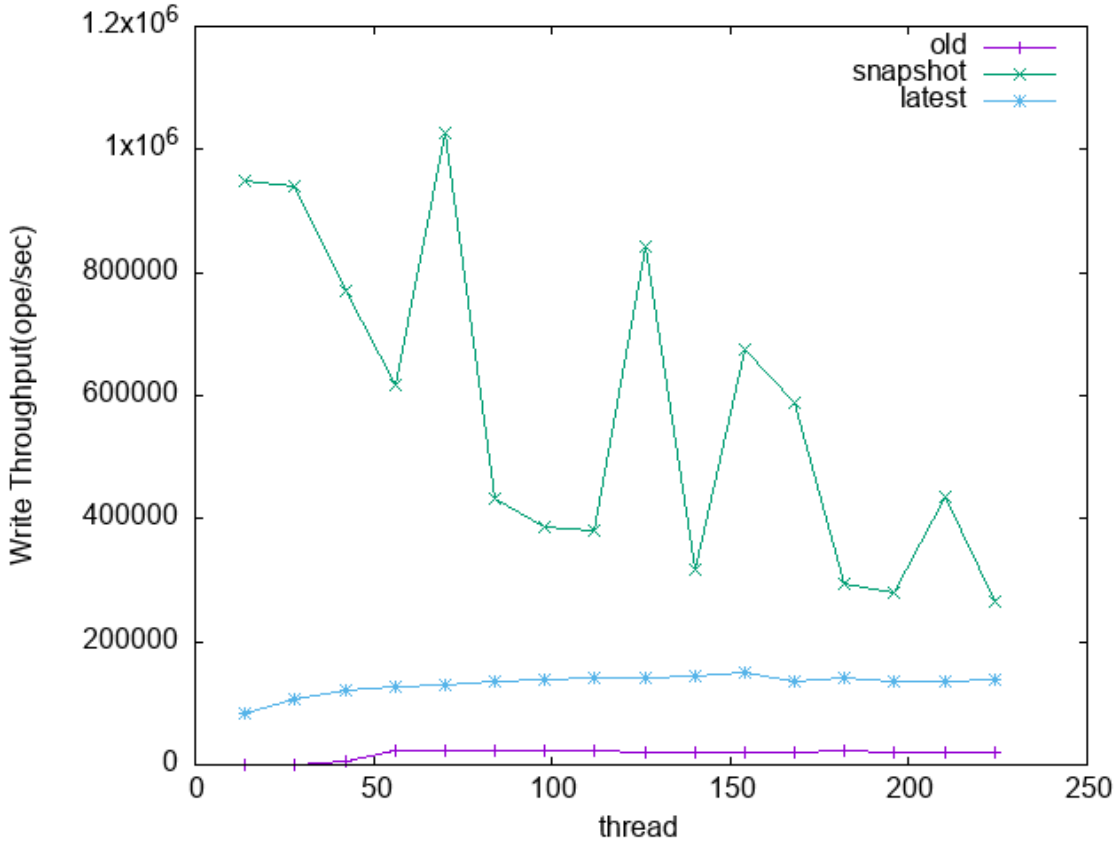

図 5.18: opposite_write フラグを有効にした場合

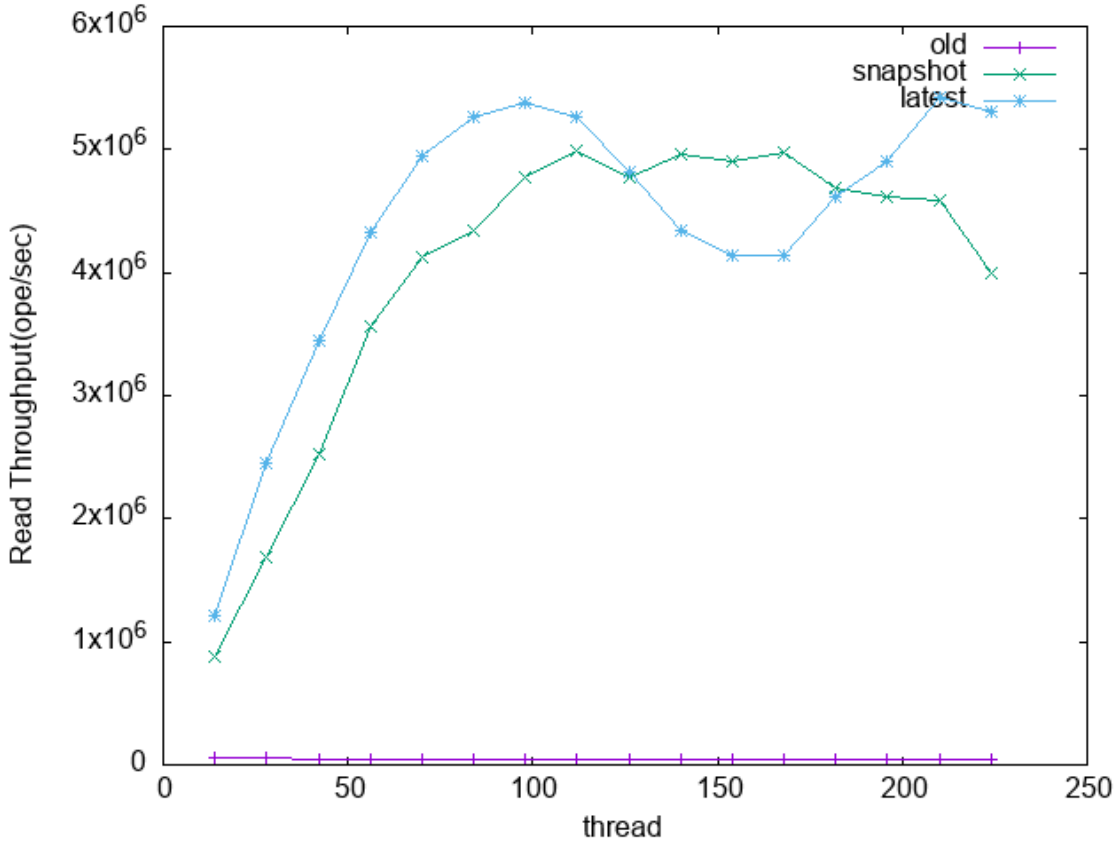

図 5.19: opposite_write フラグを無効にした場合

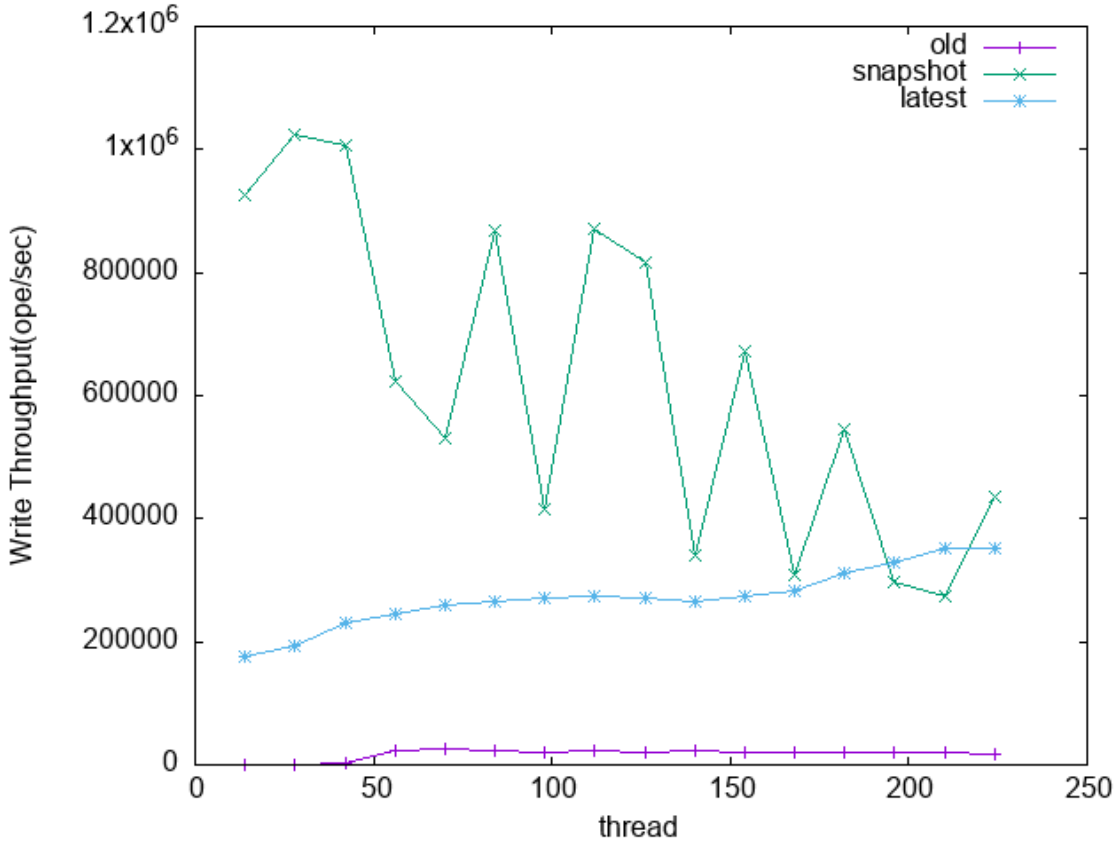

図 5.20: opposite_write フラグを無効にした場合

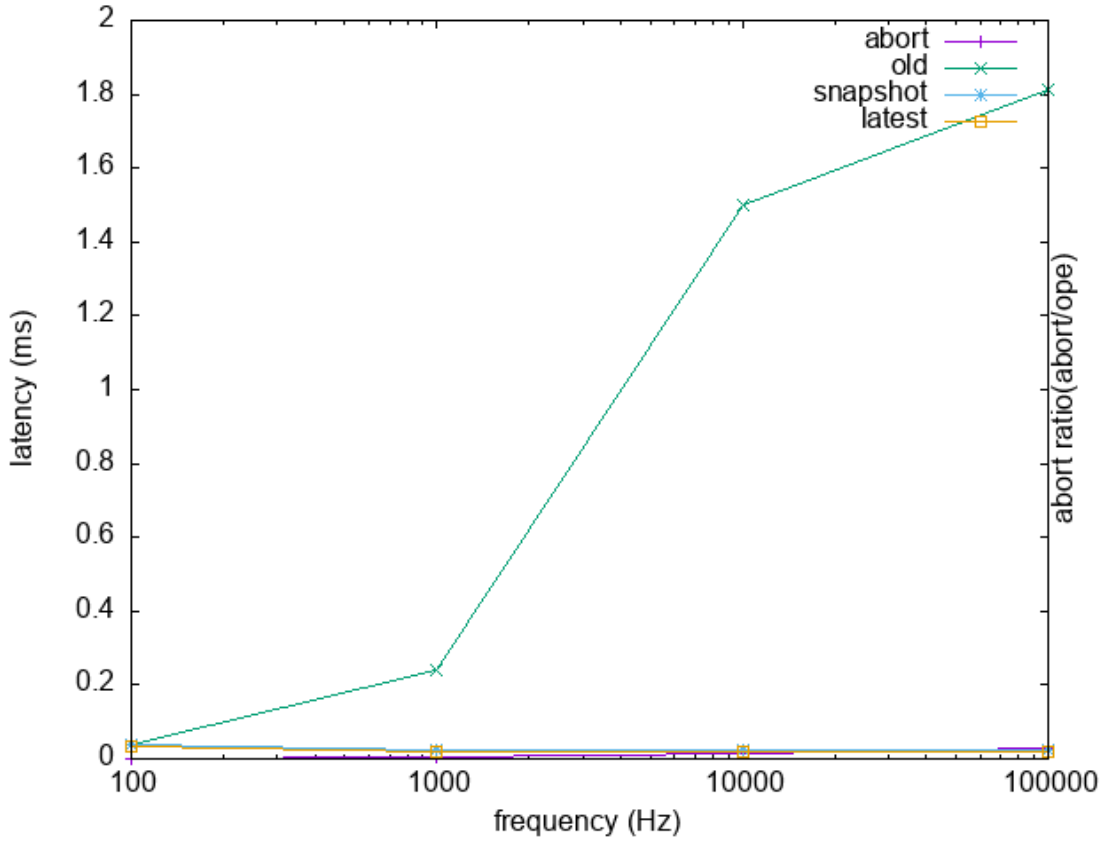

図 5.21: 制御周期とレイテンシ

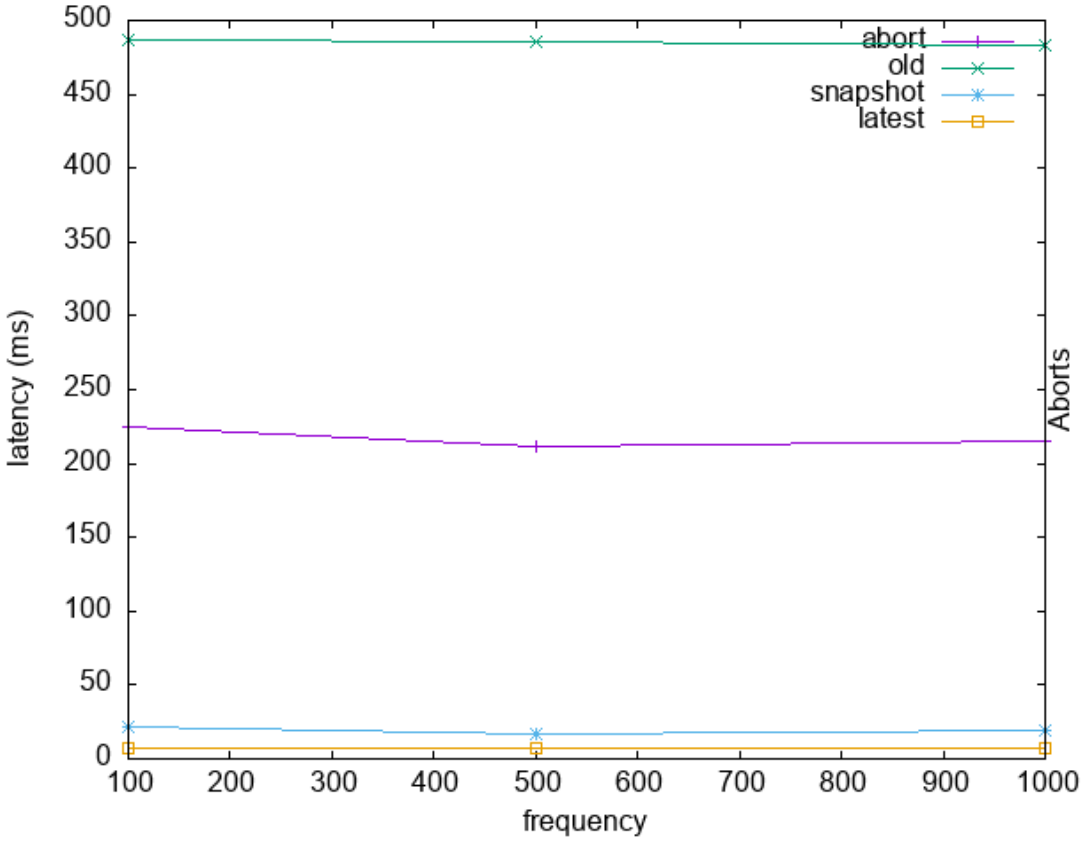

図 5.22: 制御周期とレイテンシ

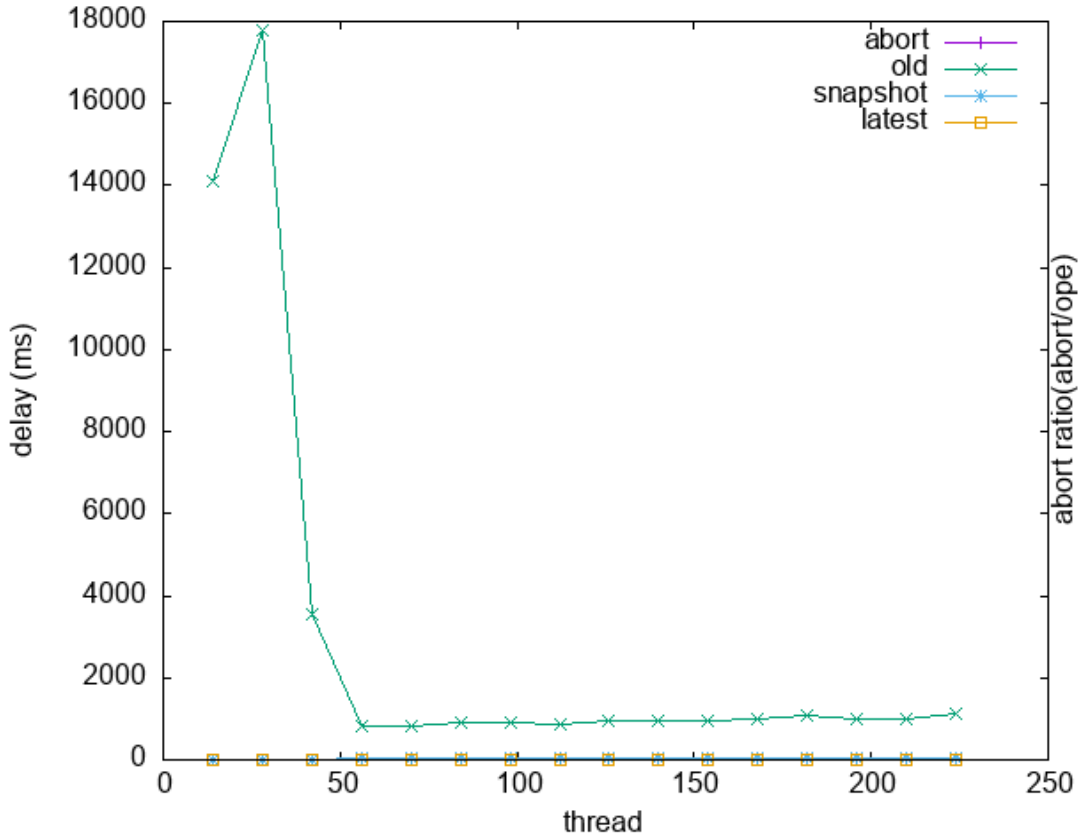

図 5.23: YCSB-A スレッド数と delay

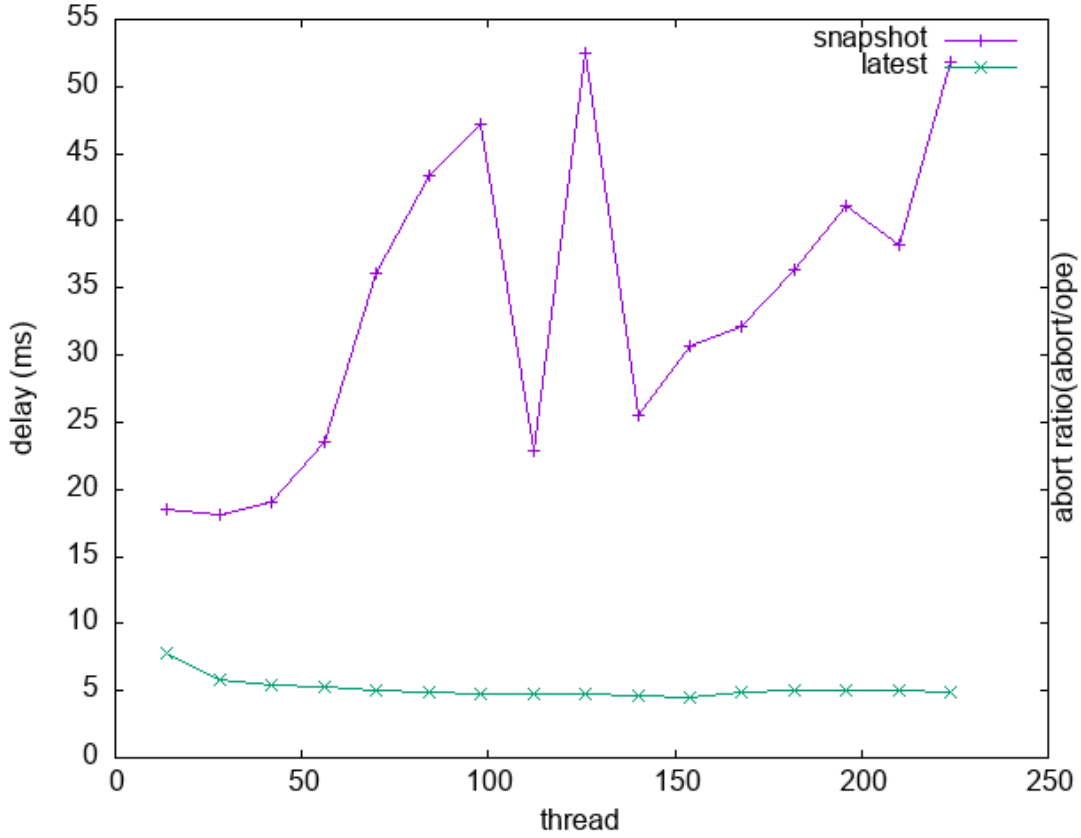

図 5.24: YCSB-A スレッド数と delay snapshot と latest のみ

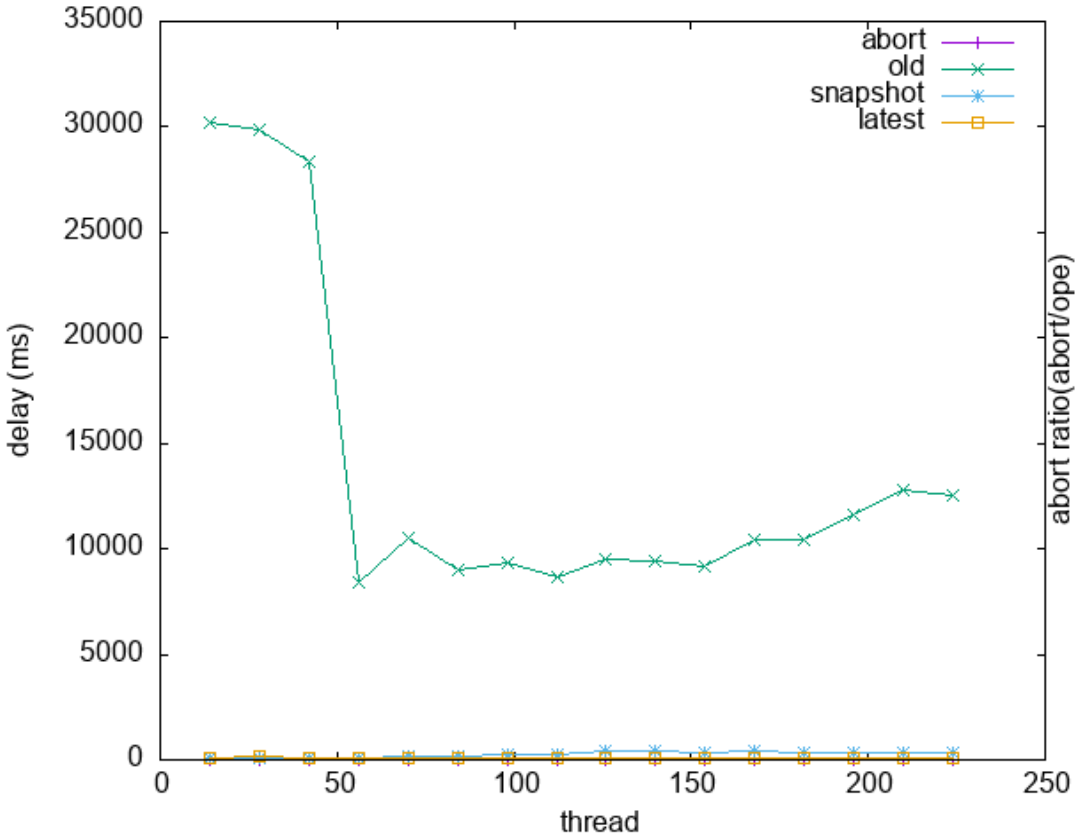

図 5.25: YCSB-B スレッド数と delay

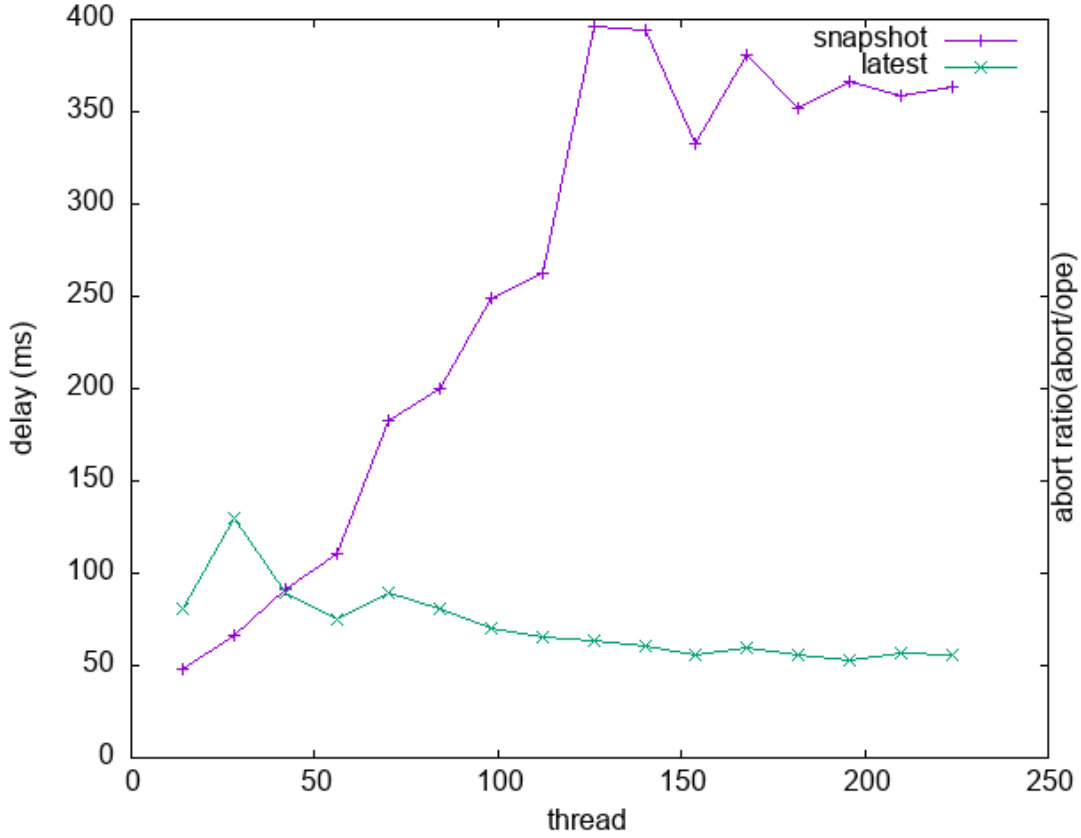

図 5.26: YCSB-B スレッド数と delay

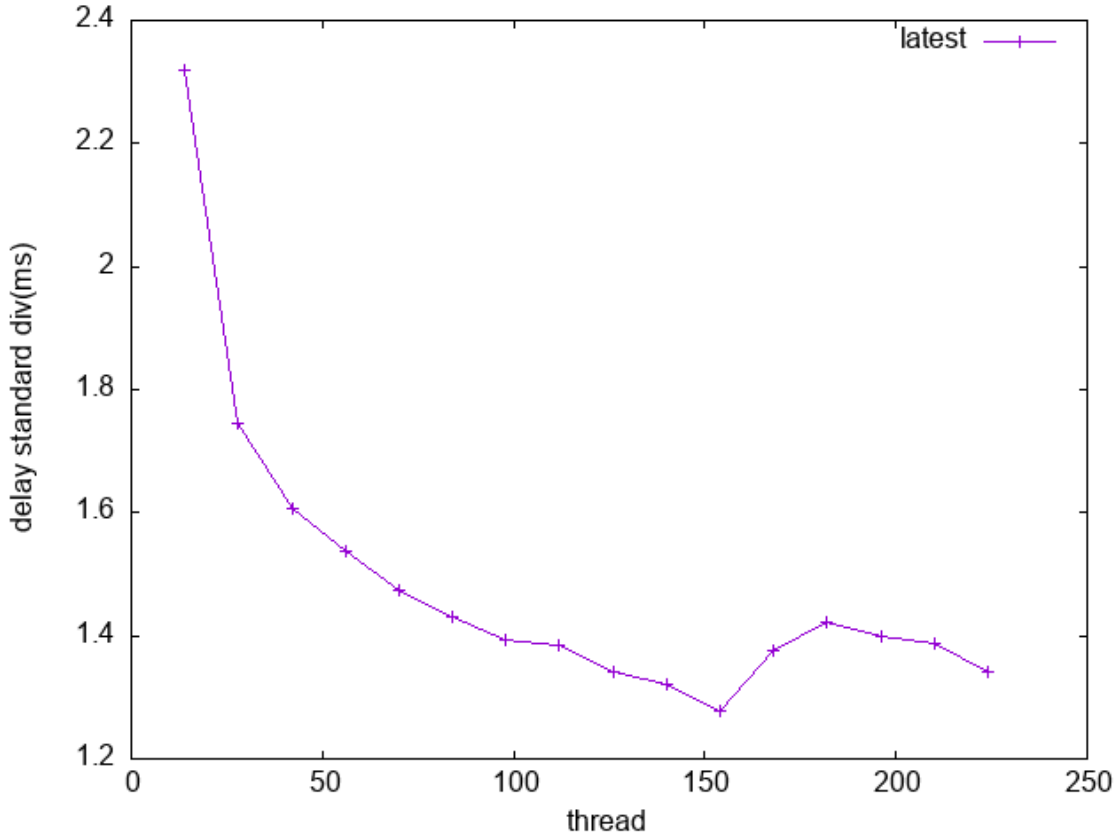

図 5.27: YCSB-A スレッド数とデータの同一性

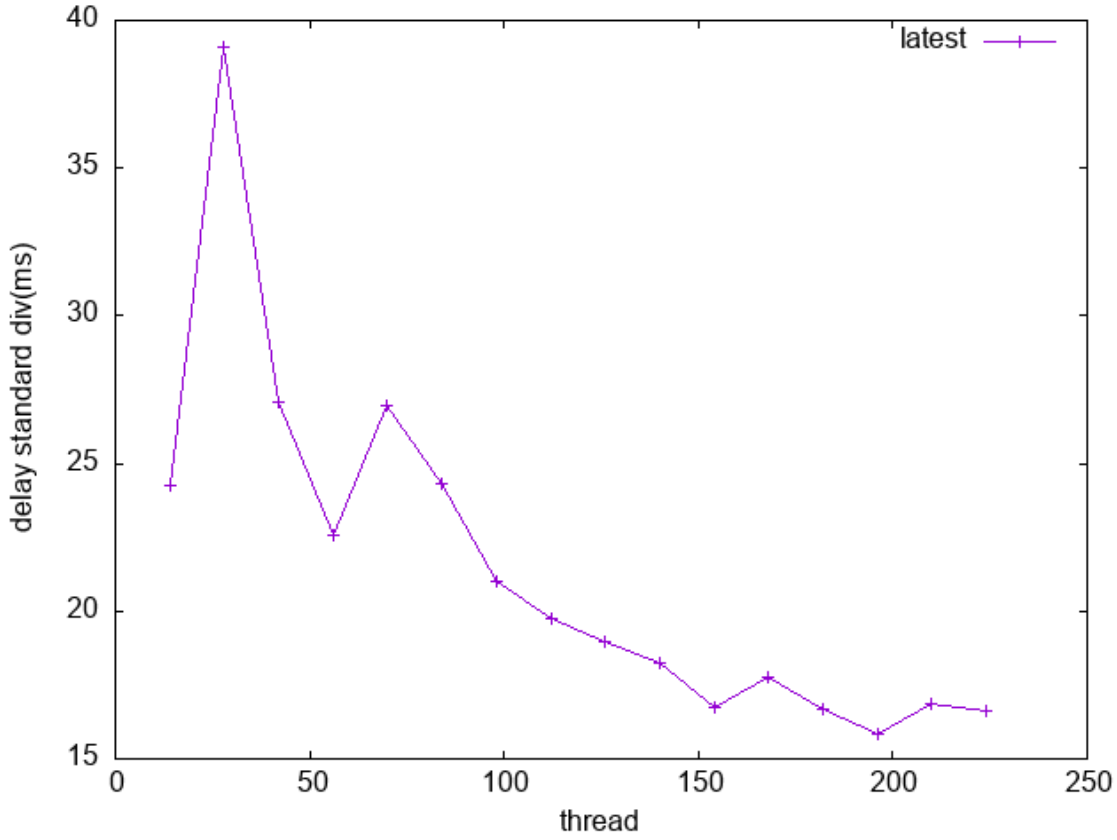

図 5.28: YCSB-B スレッド数とデータの同一性

第6章　結論

再粒度ロックの導入により、パフォーマンスを上げることができた特に、スレッド数の増加とともにスループットが下がる問題を解決できた

データの一貫性は必須、というデータも示したい。

を明らかにした

ヘビ型ロボットや、60 秒間のデータを取ろう！

また latest は snapshot より高いスループット、レイテンシーを記録し、多くのパターンにおいてよりデータの鮮度が高い

なるべく最新のデータを元にこのようなワークロードにおいては

第7章　今後の課題

ここでは取り上げなかった TF 木の問題点として、部分的に座標情報の登録に失敗するケース。rollback などもいれ transactional に行う必要
また、insert/delete が大量に発生するケースも考える。
Silo の実装
優先順位キューの実装も検討

謝辞

本研究を進めるにあたり、慶應義塾大学准教授川島英之先生、筑波大学システム情報系情報工学域大矢晃久、筑波大学システム情報系情報工学域萬礼応先生に頂きました優れた御指導により、私の研究はとても有意義で満ち足りたものとなりました。また、慶應義塾大学川島研究会秘書藤川綾様には幾多の手続きを丁寧にサポートして頂き，円滑な出張や書類作成，研究環境整備を行うことができました。この研究に関わっていただいたすべての方に深く感謝を申し上げます。

参考文献

[1] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in ICRA Workshop on Open Source Software, 2009.

[2] T. Foote, "tf: The transform library," 2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA), 2013, pp. 1-6, doi: 10.1109/TePRA.2013.6556373.

[3] Philip A. Bernstein and Nathan Goodman, "Concurrency Control in Distributed Database Systems" in ACM Computing Surveys, 1981, pp. 185-221

[4] "BufferCore.h", https://github.com/ros/geometry2/blob/noetic-devel/tf2/include/tf2/buffer_core.h

[5] "GAIA platform", https://www.gaiaplatform.io

[6] "ROS2", https://docs.ros.org/en/rolling/

[7] "Autoware", https://tier4.jp/en/autoware/

[8] Stephen Tu, Wenting Zheng, Eddie Kohler †, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In SOSP, pages 18–32. ACM, 2013

[9] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. InProceedings of the 2017 ACM International Conference onManagement of Data, p. p. 21âĂŞ35, 2017.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. InProceedings of the 1st ACM symposiumon Cloud computing, p. p. 143âĂŞ154, 2010.

[11] Guna Prasaad, Alvin Cheung and Dan Suciu. Improving High Contention OLTP Performance via Transaction Scheduling.