環境情報学部論文

並行性制御法によるROS TFの高品質化

2022年1月

71970013 / t19501yo
荻原　湧志

並行性制御法による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

並行性制御法による 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、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。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が完全に逐次化され、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その仕様によって最新のデータを参照し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並行性制御技術における 細粒度ロッキング法、及び 2PL を応用することにより、これら問題を解決した。提案手法では既存手法と比べ最大 257 倍のスループット、最大 282 倍高速化したレイテンシ、最大 132 倍のデータ鮮度となることを示した。

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 [1]。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。この位置関係は平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 [1]。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。

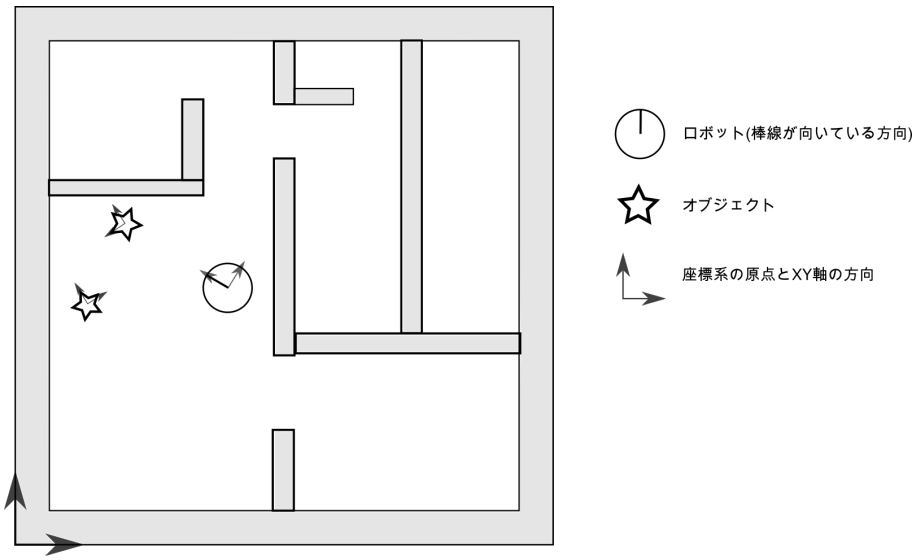

図 1.1: 部屋の中のロボット

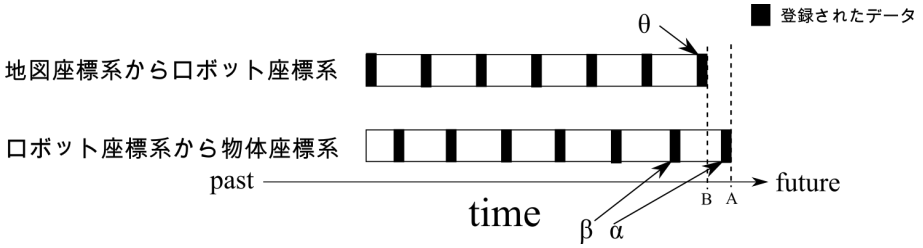

図 1.2: 位置関係の登録のタイムライン

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 [1]。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 [1]。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 [1]。

1.2 TF ライブラリ

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 [1]。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。

上述したように、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。

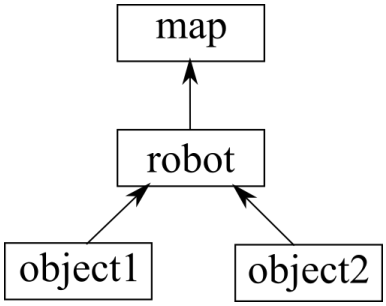

図 1.3: 図1.1に対応する木構造

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。

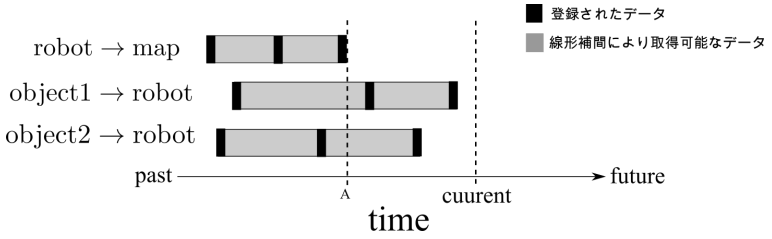

図 1.4: 図1.3における位置関係登録のタイムライン

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。

1.3 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同時に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スループットやレイテンシに大きな問題を与える。

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。

1.4 貢献

上述した問題 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することによって解決した。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。これにより、既存手法ではジャイアントロックによって TF へのアクセスは完全に逐次化されていたが、提案手法では細粒度ロックによって並行にアクセスができるようになった。細粒度ロックを実装した場合のスループットは最大で 11,574,200、レイテンシは高々0.7ms となった。また既存手法と比べ細粒度ロックを実装した場合はスループットは最大 243 倍、レイテンシは最大 172 倍高速化した。

上述した問題 2 については、データベースの並行性制御技術における 2PL を適用することにより、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 (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) を提供することによって解決した。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、複数のデータ操作を atomic に行えるようにする手法である。これにより、複数のデータの最新の座標変換情報の読み込み・書き込みを効率的に atomic に行えるようになった。既存手法で提供されているインターフェイスではジャイアントロックにより TF へのアクセスが逐次化され、また§1.2にて説明したように時刻の同期をとるという仕様のために過去のデータを参照しデータの鮮度が落ちるという問題があったが、lookupLatestTransform と setTransforms を使うことによりこれは解決できた。既存手法と比べ、lookupLatestTransform と setTransforms を使った場合にはデータの鮮度は最大で 132 倍となった。また、既存手法と比べ最大 257 倍のスループット、最大 282 倍高速化したレイテンシとなり、このインターフェイスはスループットとレイテンシにおいても既存手法より優れていることを示した。

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と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。

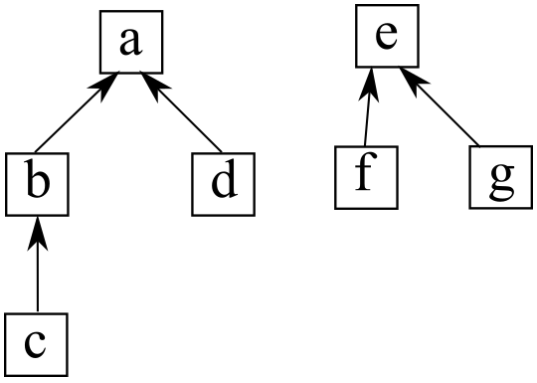

図 3.1: 複数の木構造

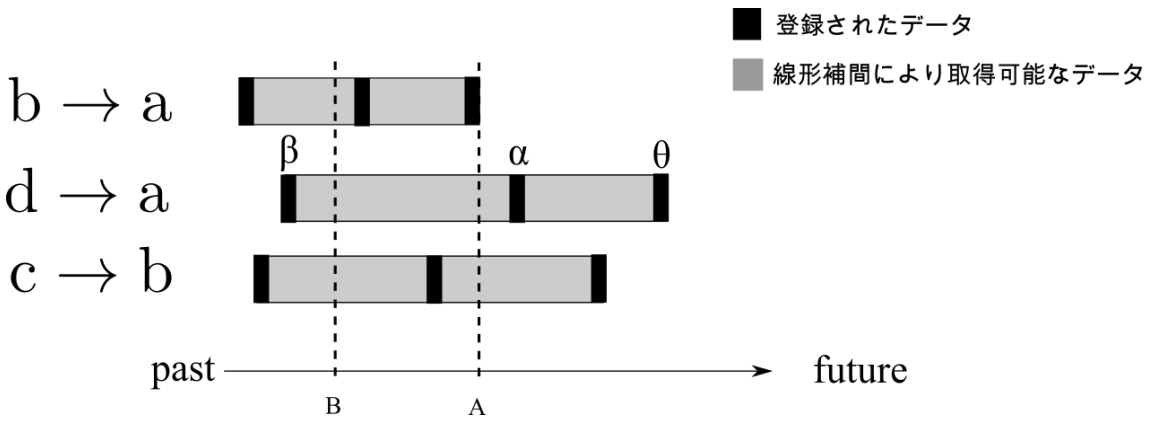

図 3.2: タイムライン

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
21:    return source_trans * (target_trans)^-1
22: 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 for
28:    return common_time
29: end function
```

Algorithm 3 setTransform

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

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、森構造へのアクセスは完全に逐次化され、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。

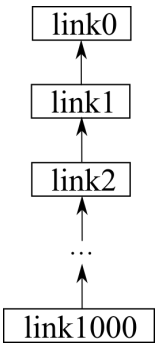

図 3.3: ヘビ型ロボットにおける森構造

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 2PL を適用し、これらの問題を解決する。

4.1　細粒度ロックの導入

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。

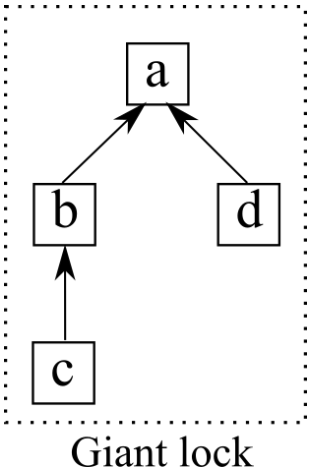

図 4.1: Giant lock

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。

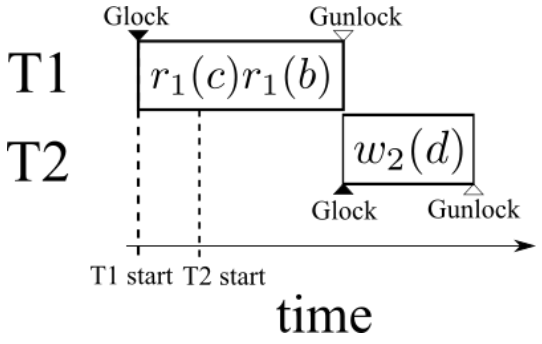

図 4.2: ジャイアントロックにおけるスケジュール

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: 各ロックの互換性

このルールは、表4.1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。

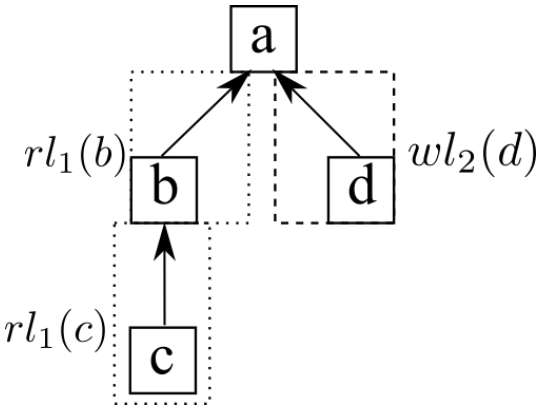

図 4.3: 細粒度ロッキング

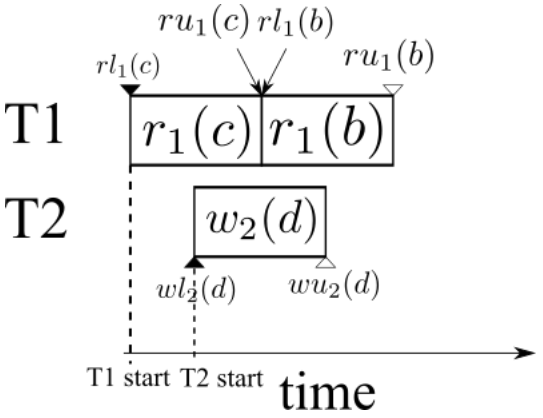

図 4.4: 細粒度ロックにおけるスケジュール

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。

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4.1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。

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
25:     return source_trans * (target_trans)^-1
26: 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 for
32:     return common_time
33: end function
```

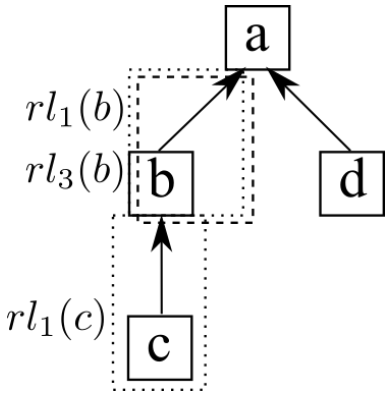

図 4.5: 二つの読み込みロック

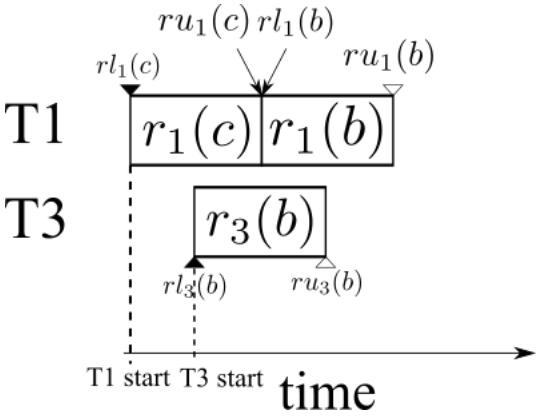

図 4.6: 二つの読み込みロックにおけるスケジュール

4.2 2PL の導入

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。

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。

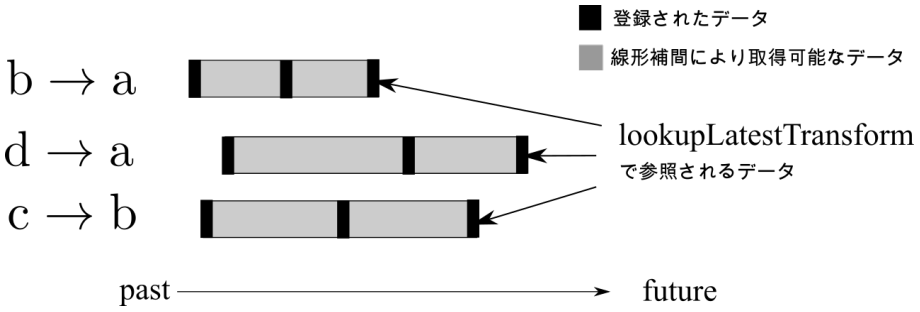

図 4.7: lookupLatestTransform で取得するデータ

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。

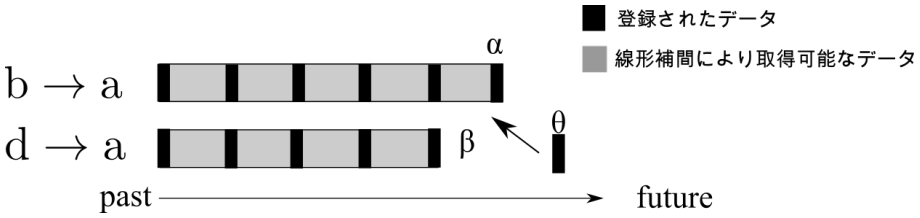

図 4.8: 同時に座標変換が登録されるケース

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並行性制御技術である。このような性質は、並行性制御技術においては 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。

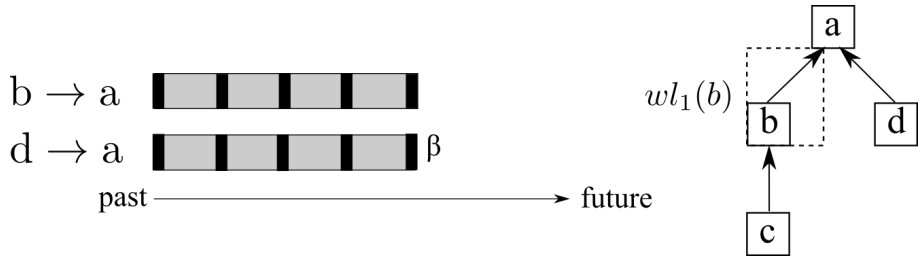

図 4.9: setTransforms と lookupLatestTransform 1

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。

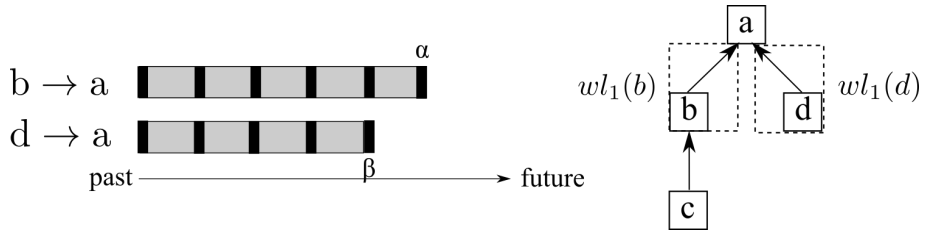

図 4.10: setTransforms と lookupLatestTransform 2

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 2 は待機する必要がある。

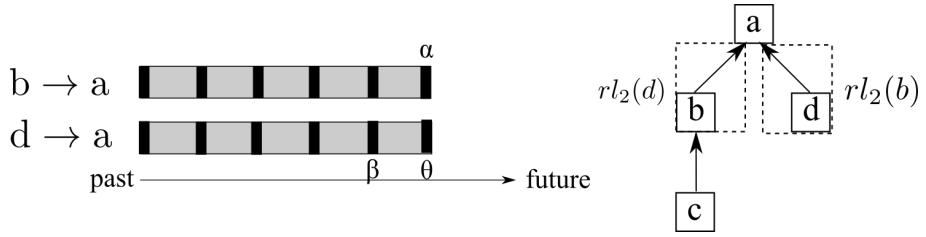

図 4.11: setTransforms と lookupLatestTransform 3

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

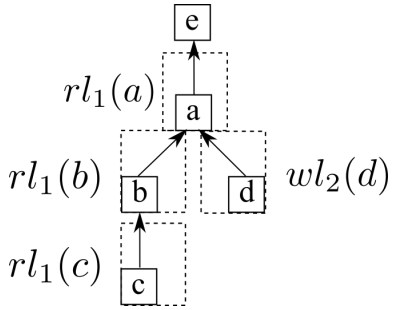

図 4.12: deadlock

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。

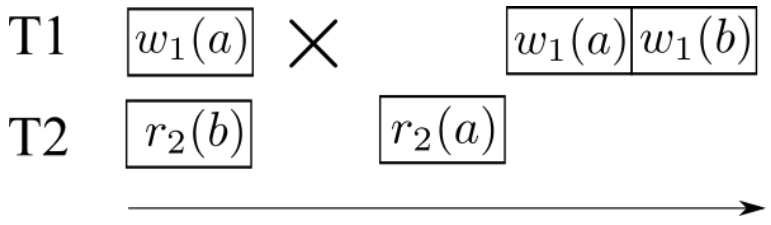

図 4.13: Dirty Read

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。図中の × 印は全てのロックを外し、処理を最初からやり直す事を表している。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。

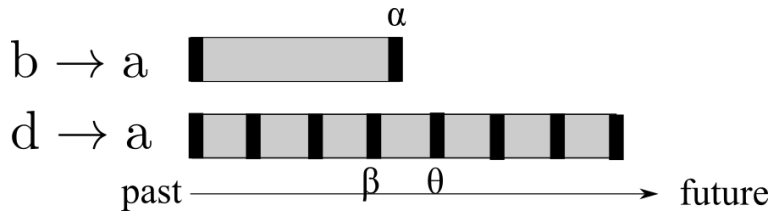

図 4.14: deadlock

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、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
22:     return source_trans * (target_trans)^-1
23: end function
```

Algorithm 7 setTransforms

```
1:  procedure 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 procedure
```

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を表す。アクセスした各座標変換データのタイムスタンプの平均とアクセス時の時刻の差を delay とし、delay が少ない方がデータの鮮度が高いとみなす。
4. データの同期性 (lookupLatestTransform のみ): 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を表す。アクセスした各座標変換データのタイムスタンプの標準偏差を計算し、これが小さい方がデータの同期性があるとみなす。
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。

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=1000000、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5.1のように、old に比べて snapshot は最大 243 倍、latest は最大 257 倍のスループットとなった。
レイテンシについては図5.2、図5.3のように、どの手法においてもレイテンシとスレッド数が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提案手法が既存手法より優れているのは、既存手法ではジャイアントロックにより操作を並行に行えないが、提案手法では細粒度ロックと 2PL によって操作を並行に行えるからだと考えられる。また、latest の方が snapshot より優れた性能をしてしているのは、§4.1で説明したように snapshot の lookupTransform では、森構造を 2 度読み込む必要があるからだと考えられる。
書き込みが発生しないため、YCSB-C における書き込みスループット、書き込みレイテンシ、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。

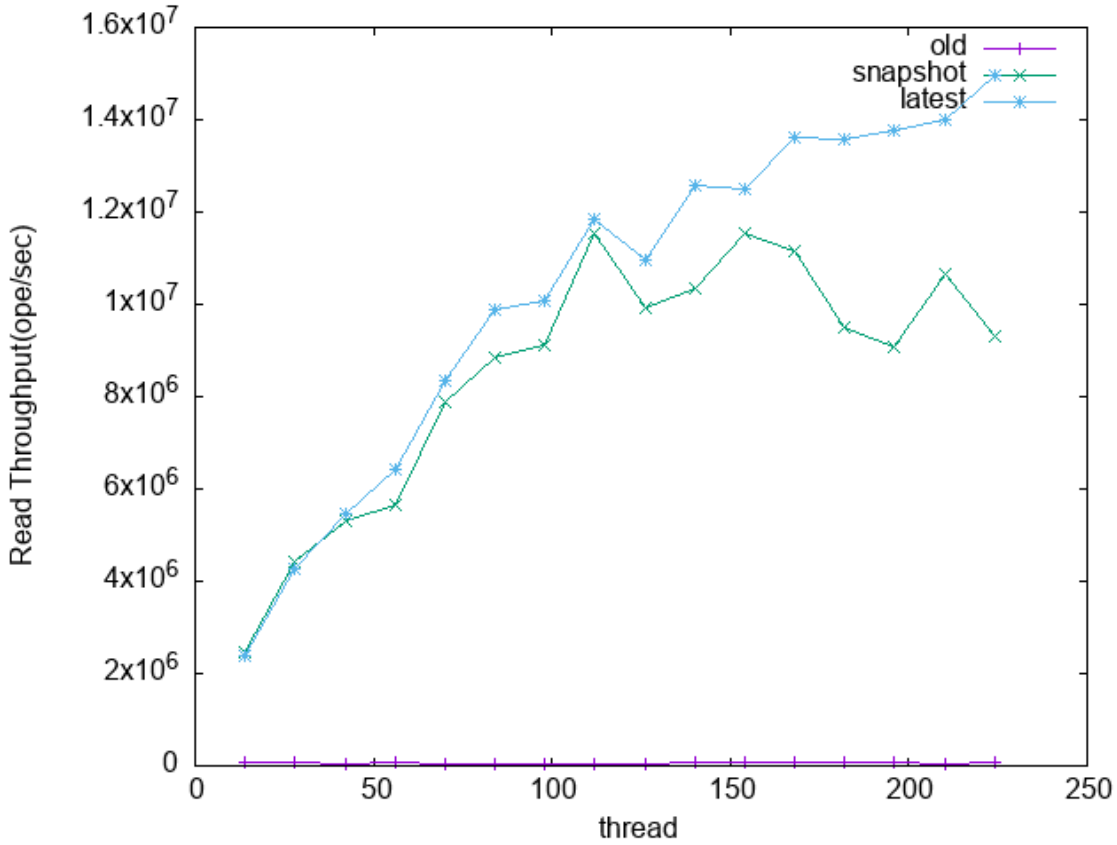

図 5.1: YCSB-C におけるスレッド数と読み込みスループットの関係

5.4 YCSB-A

YCSB-A ではスループットについては図5.4のように old に比べて snapshot は最大 61 倍、latest は最大 143 倍のスループットとなった。また、スループットを読み込みスループット、書き込みスループットと分けてそれぞれ図5.5、図5.6で表示した。ここで、snapshot については読み込みスループットについてはある一定以上のスレッド数では性能が低下し、また書き込みスループットについてはスレッド数の増加とともに性能が低下していることがわかる。

このようになる原因を調べるため、スレッド数とキャッシュミス率の関係について調べ、図5.7に示した。ここからわかるように、latest ではキャッシュミス率がほとんど変化しないのに対し、snapshot ではキャッシュミス率が上がっていることがわかる。§ 4.1で説明したように snapshot では lookupTransform にて同じ要素に対して二回の読み込みがあル。この一回目の読み込みと二回目の読み込みの間にて setTransform による同じ要素への書き込みが発生するとキャッシュが汚染され、二回目の読み込み時にキャッシュミスとなる。スレッド数の増加とともにこの現象が増えることがキャッシュミス率の増加に繋がり、読み込みスループットと書き込みスループットの性能低下につながったと考えられる。

読み込み・書き込み専用スレッドそれぞれのレイテンシについて図5.8〜5.9に表示した。図5.10のように old の書き込みレイテンシがスレッド数が少ない状況にて非常に悪いパフォーマンスを示しているのは、old ではジャイアントロックにより操作を逐次的にしか行えないからだと考えられる。また、図5.9、図5.11のように snapshot の読み込み、書き込みレイテンシがスレッド数とともに latest より増加しているのは、上述したようにキャッシュミスの

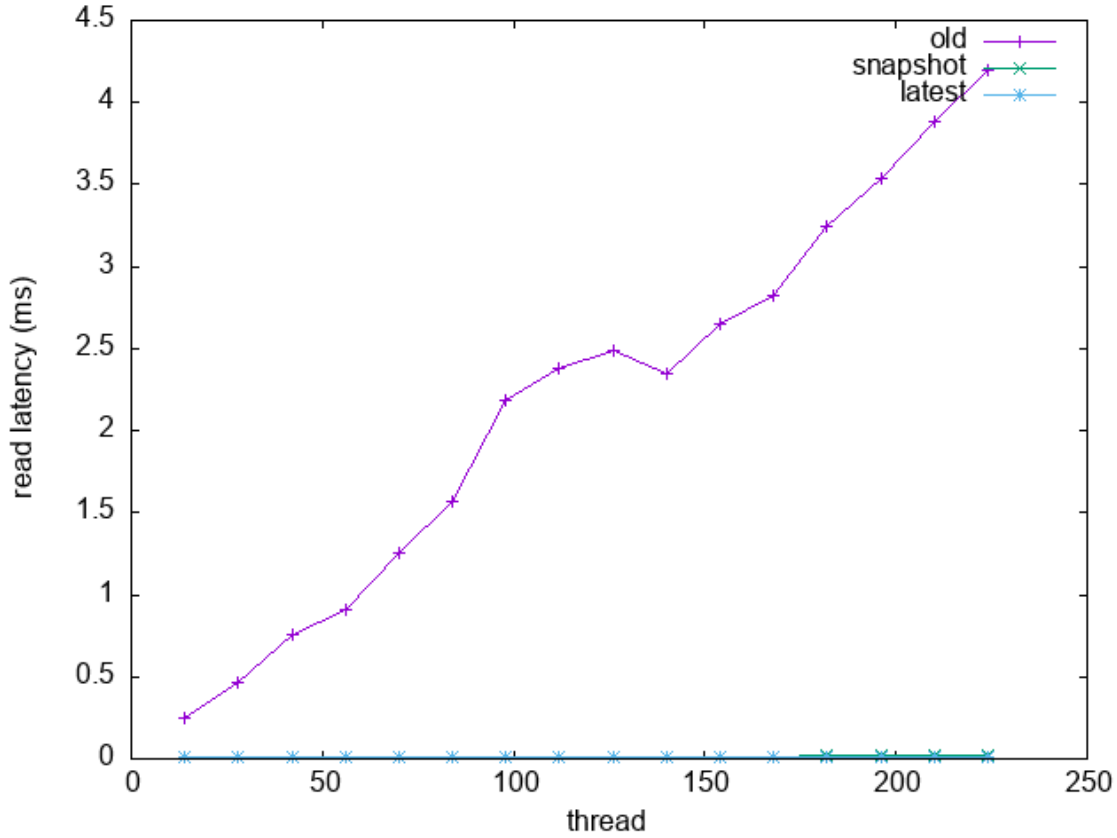

図 5.2: YCSB-C におけるスレッド数とレイテンシの関係

増加によるものだと考えられる。

スレッド数と abort 率の関係について図5.12に示した。ここからわかるように、スレッド数と abort 率が線形比例していることがわかる。

スレッド数とデータの鮮度の関係について図5.13に示した。ここからわかるように、snapshot ではスループットの低下によりスレッド数が増えるとデータの鮮度が落ちていくが、latest ではスレッド数に関係なく安定して高い鮮度のデータが取得できることがわかる。

スレッド数とデータの同期性の関係について図5.14に示した。ここからわかるように、latest ではスレッド数に関係なくデータの同期性が安定していることがわかる。

5.5 YCSB-B

スレッド数と全体のスループット、読み込みのスループット、書き込みのスループットの関係について図5.15〜5.17に表示した。また、スレッド数とキャッシュミス率の関係について図5.18に表示した。ここからわかるように、書き込みが少ないために§5.4にて説明したようなキャッシュミス率の変化は小さくなっているため、書き込みのスループットにも大きな変化が生じなかったと考えられる。

読み込み・書き込み専用スレッドそれぞれのレイテンシについて図5.19〜5.20に表示した。図5.22をみてわかるように、snapshot の方が latest より書き込みレイテンシが優れていることがわかる。これは、latest は setTransforms にて 16 の要素の書き込み、書き込みロックをする必要があるのに対し、setTransform では一つの要素の書き込み及び書き込みロックを行

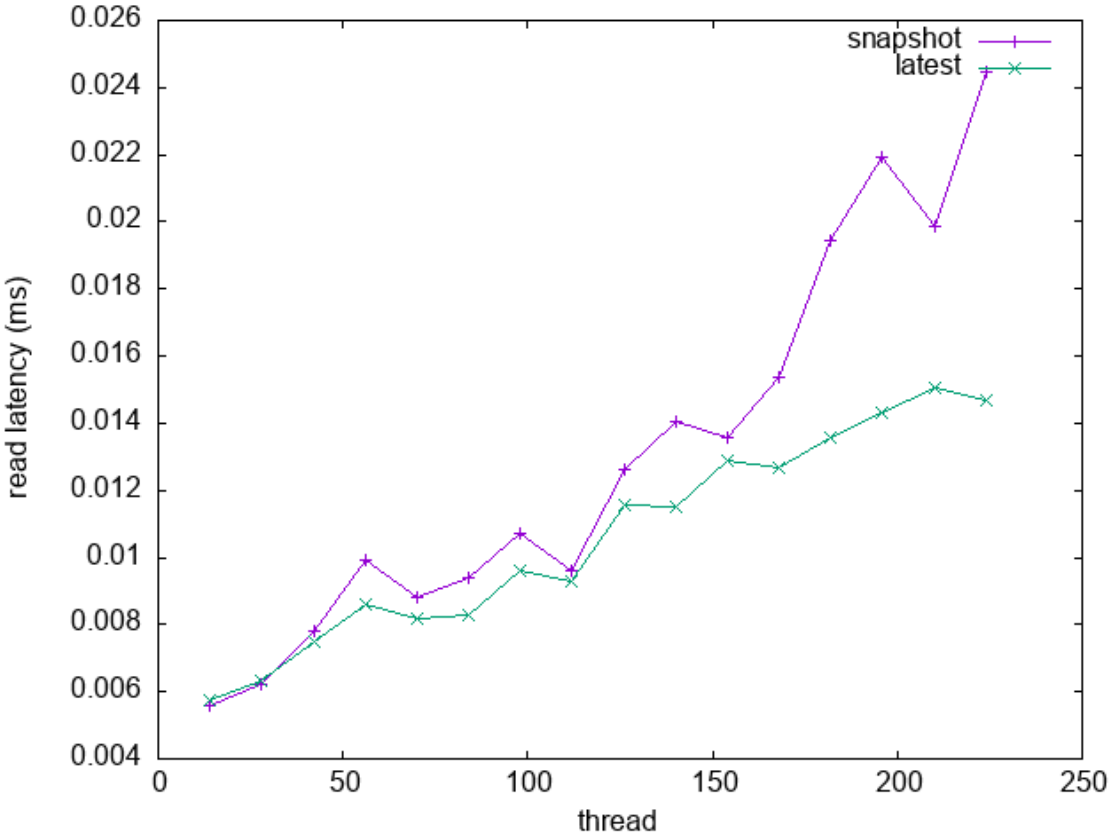

図 5.3: YCSB-C におけるスレッド数とレイテンシの関係 snapshot と latest のみ

えばよく、さらに§5.4で説明したようなキャッシュミス率の変化も発生していないからだと考えられる。

スレッド数と abort 率の関係について図5.23に示した。ここからわかるように、YCSB-B においても YCSB-A と同じようにスレッド数と abort 率が線形比例していることがわかる。

スレッド数とデータの鮮度の関係について図5.24に示した。ここからわかるように、どの手法でもスレッド数に関係なくデータの鮮度は安定しているが、latest の方が snapshot よりデータの鮮度が高いことがわかる。

スレッド数とデータの同期性の関係について図5.25に示した。ここからわかるように、latest では YCSB-A とは違いスレッド数の増加とともにデータの同期性が増加することがわかる。これは、スレッド数の増加とともに書き込みが増え、より最新のデータが増えるからだと考えられる。

5.6 制御周期

図5.26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読み込みレイテンシを表している。frequency を上げるとレイテンシも増えることがわかる。これは、制御周期を増やすことにより並行に実行される操作が増え、スレッド間の競合が増加するからだと考えられる。

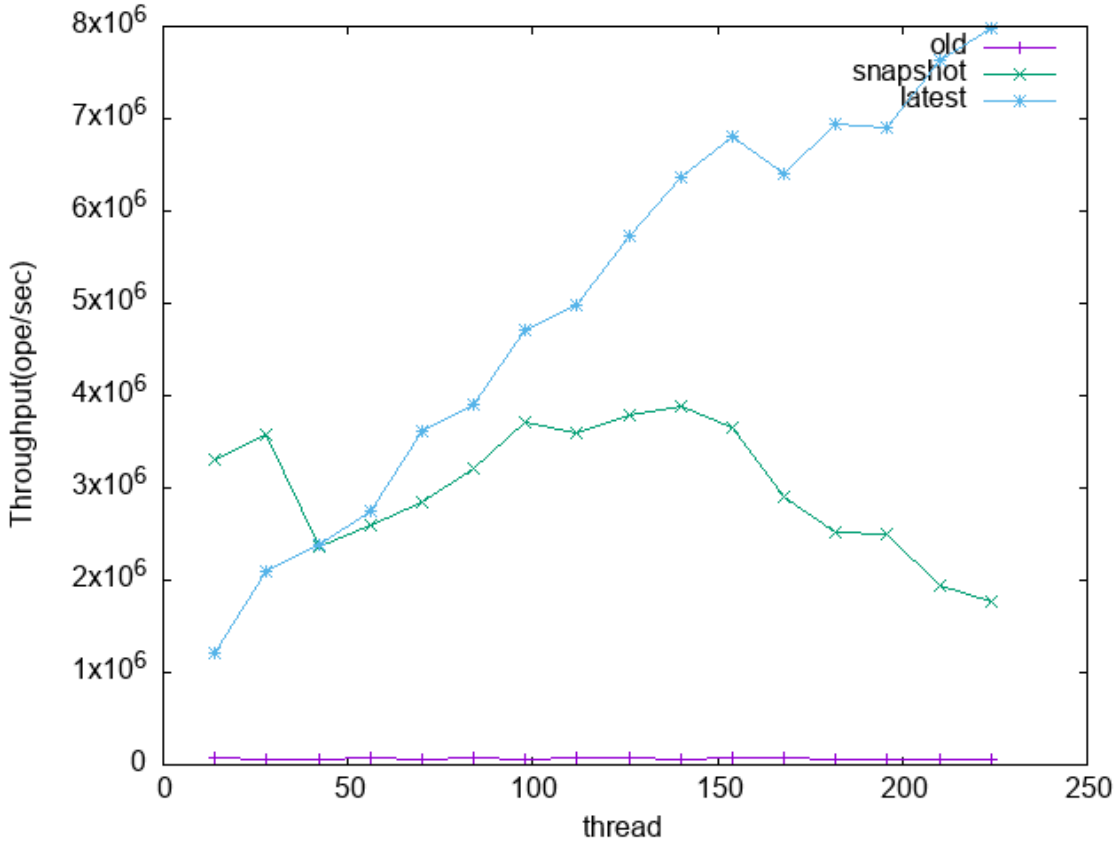

図 5.4: YCSB-A におけるスレッド数とスループットの関係

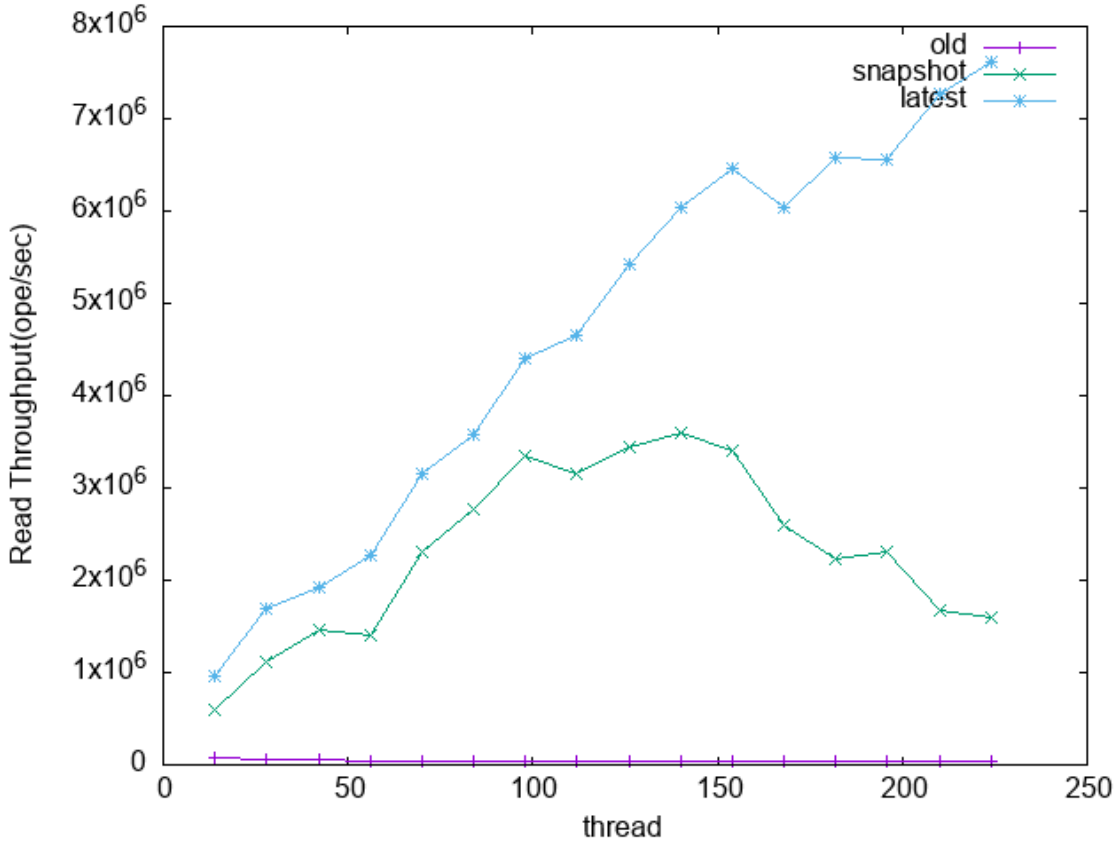

図 5.5: YCSB-A におけるスレッド数と読み込みスループットの関係

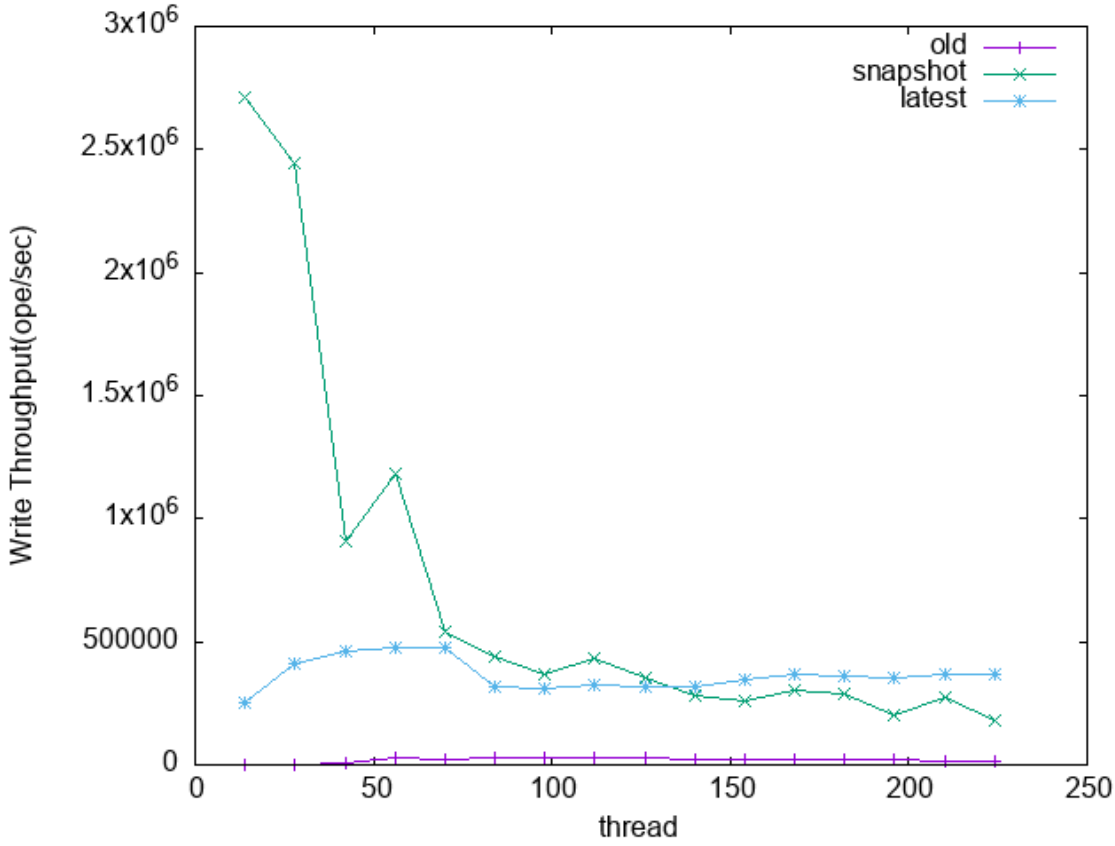

図 5.6: YCSB-A におけるスレッド数と書き込みスループットの関係

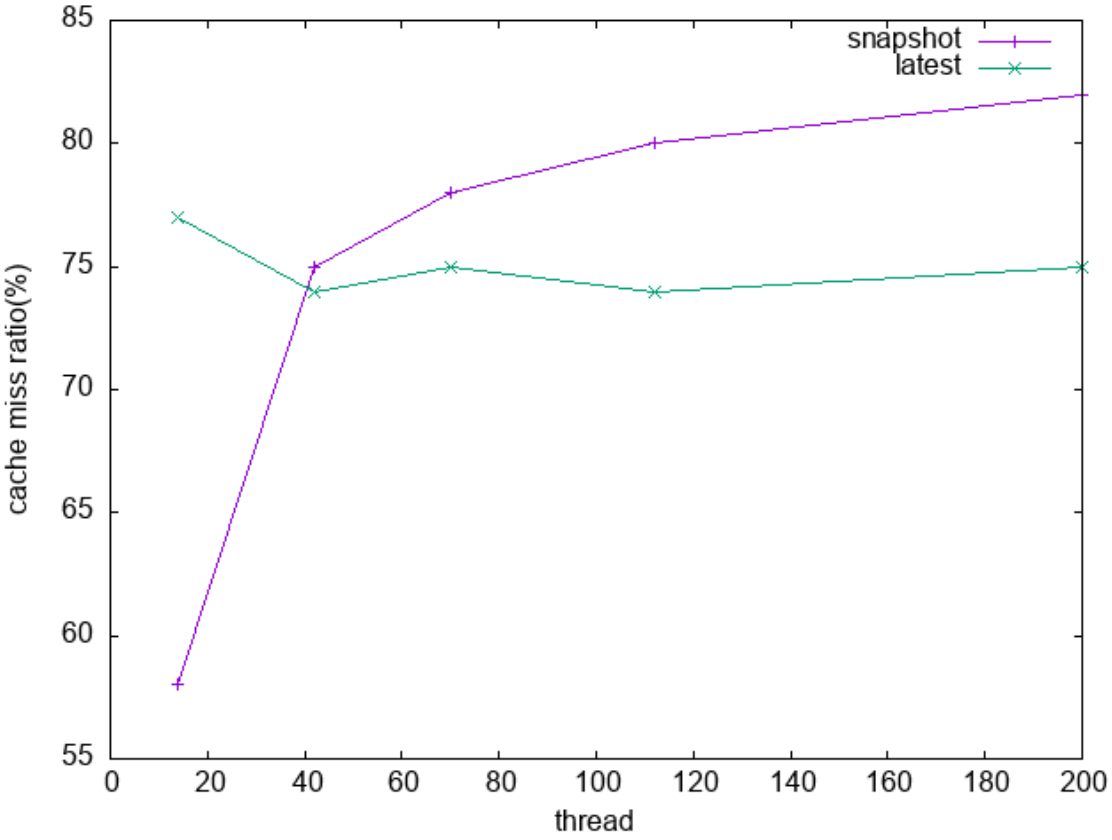

図 5.7: YCSB-A におけるスレッド数とキャッシュミス率の関係

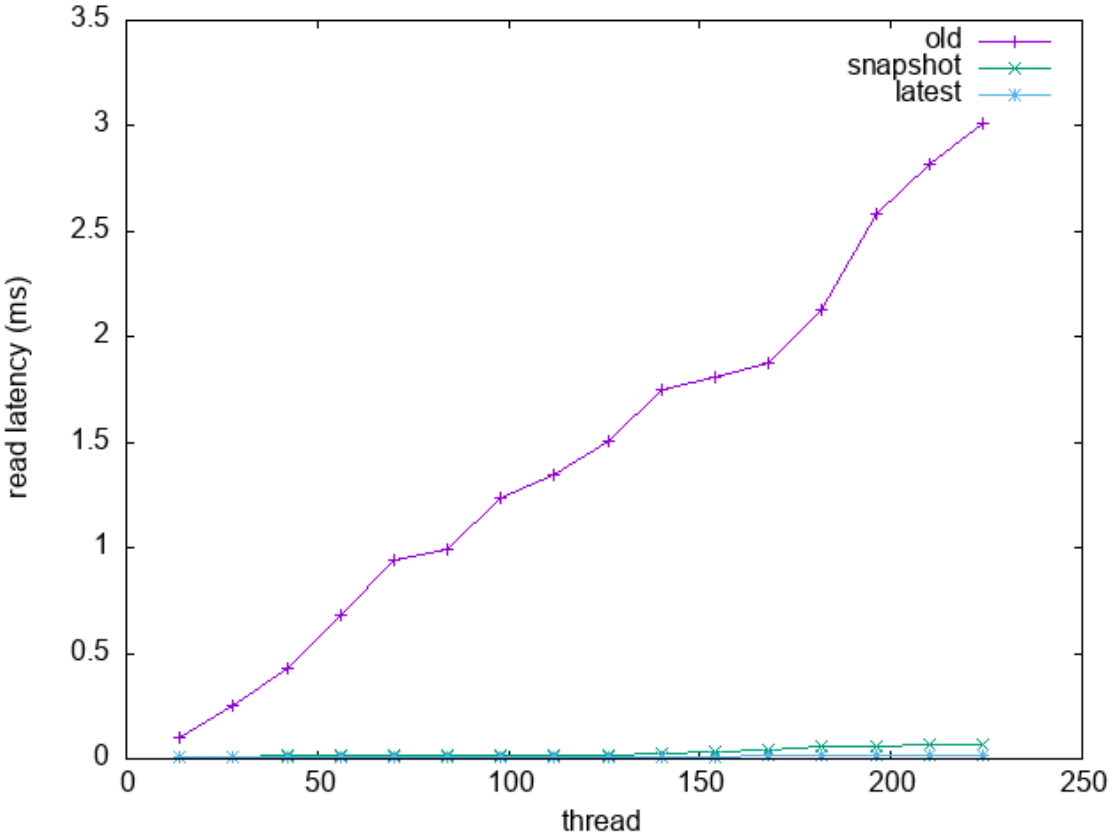

図 5.8: YCSB-A におけるスレッド数と読み込みレイテンシの関係

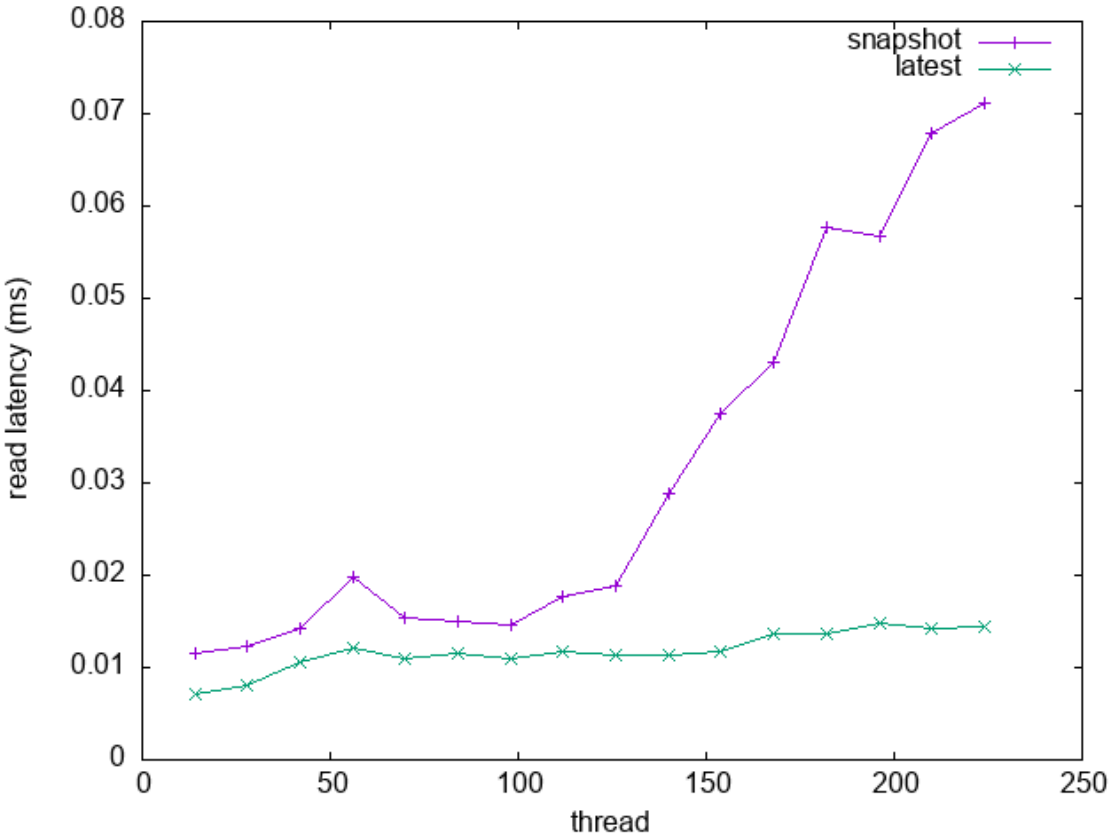

図 5.9: YCSB-A におけるスレッド数と読み込みレイテンシの関係 snapshot と latest のみ

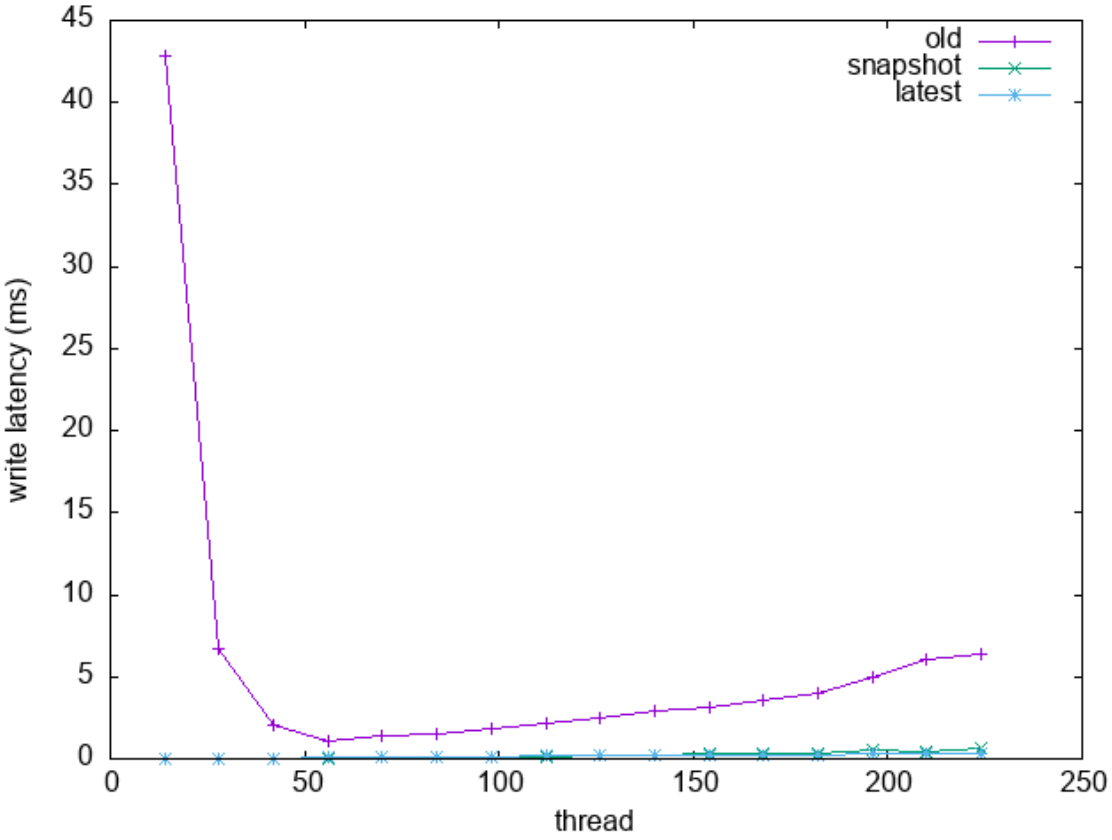

図 5.10: YCSB-A におけるスレッド数と書き込みレイテンシの関係

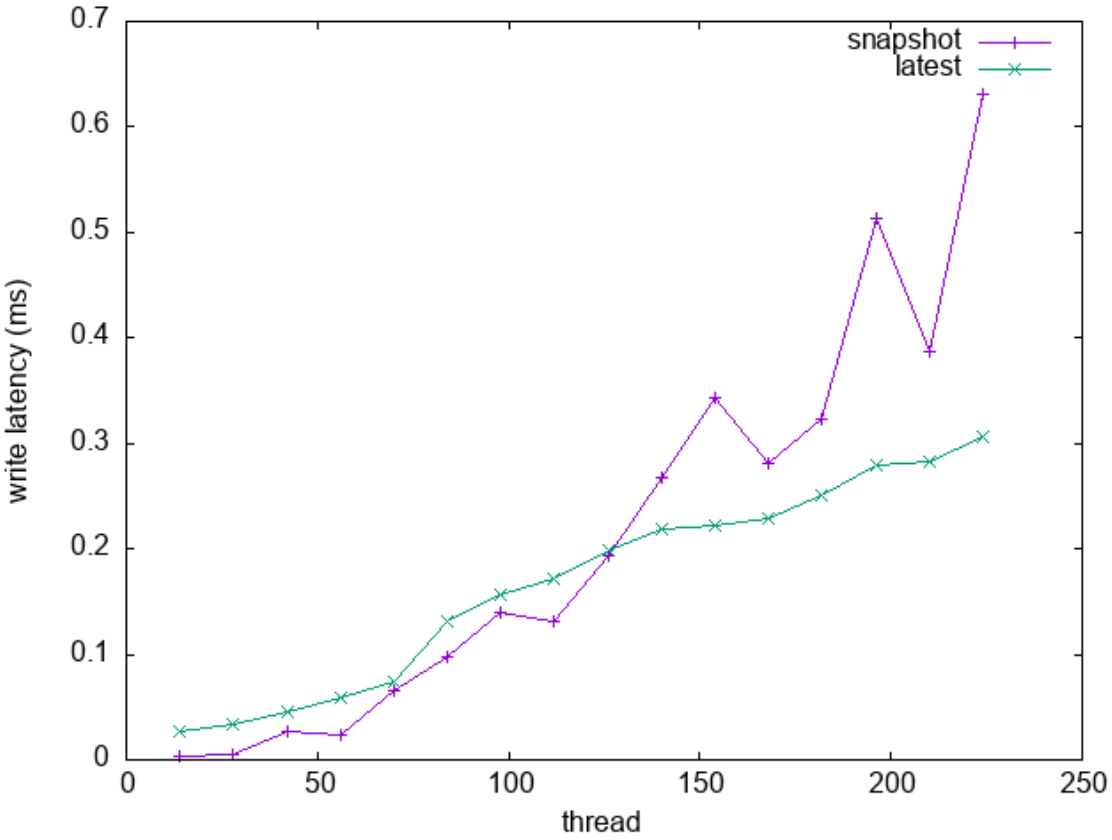

図 5.11: YCSB-A におけるスレッド数と書き込みレイテンシの関係 snapshot と latest のみ

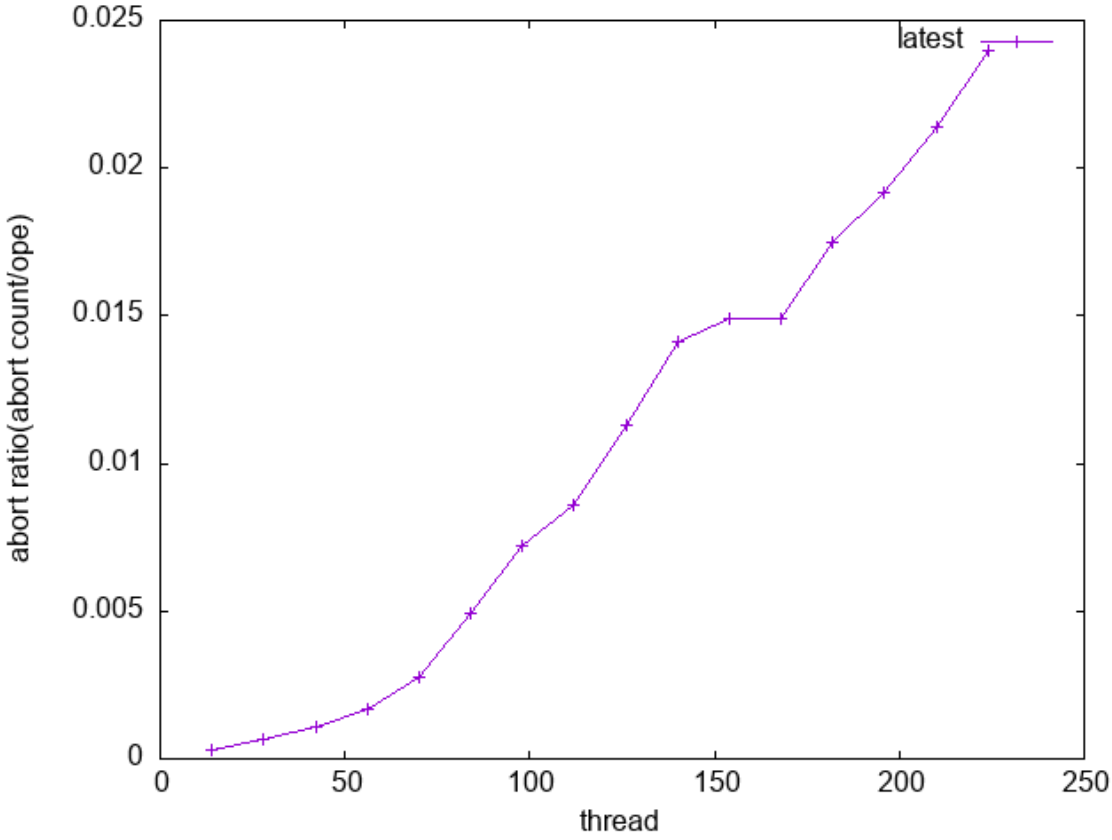

図 5.12: YCSB-A におけるスレッド数と abort 率の関係

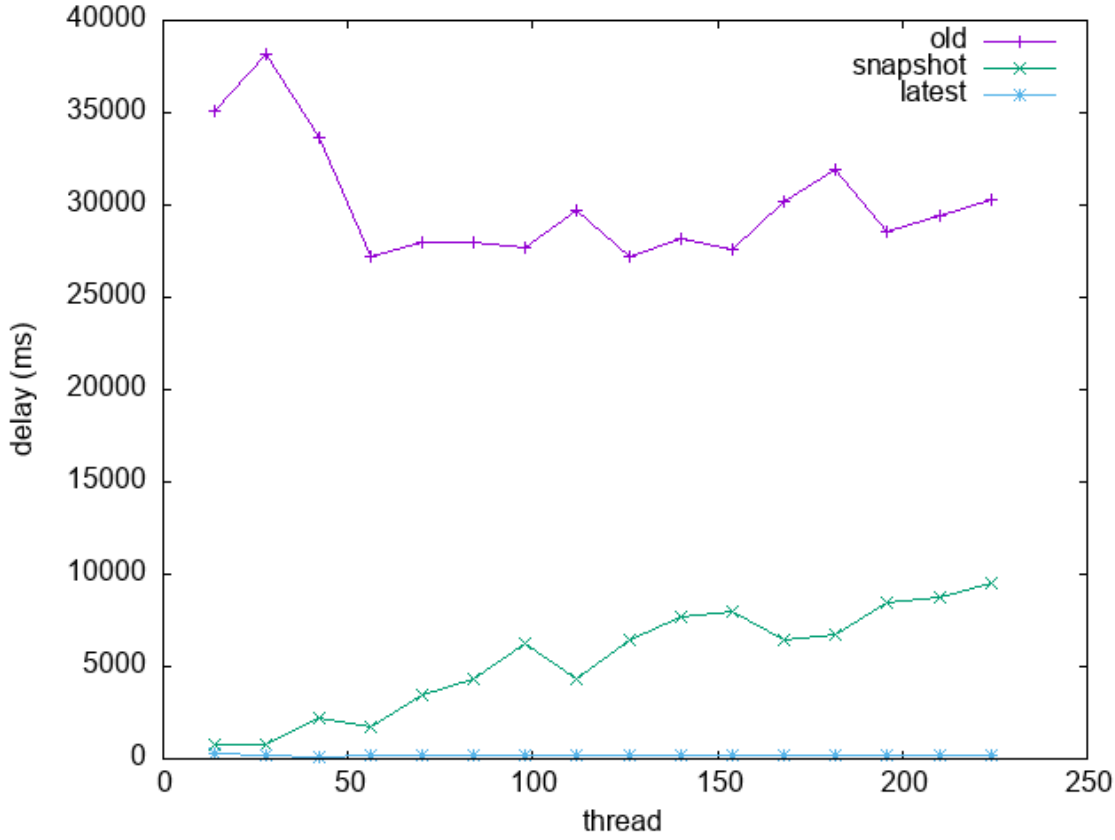

図 5.13: YCSB-A におけるスレッド数とデータの鮮度の関係

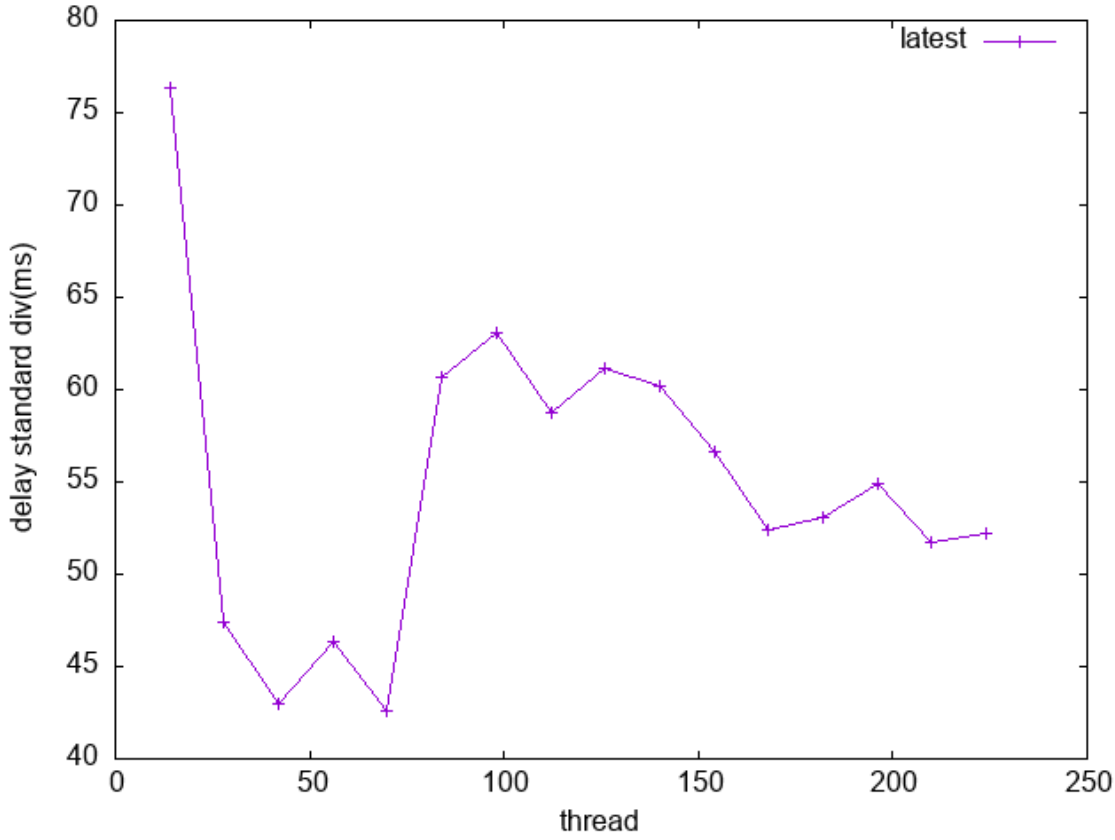

図 5.14: YCSB-A におけるスレッド数とデータの同期性の関係

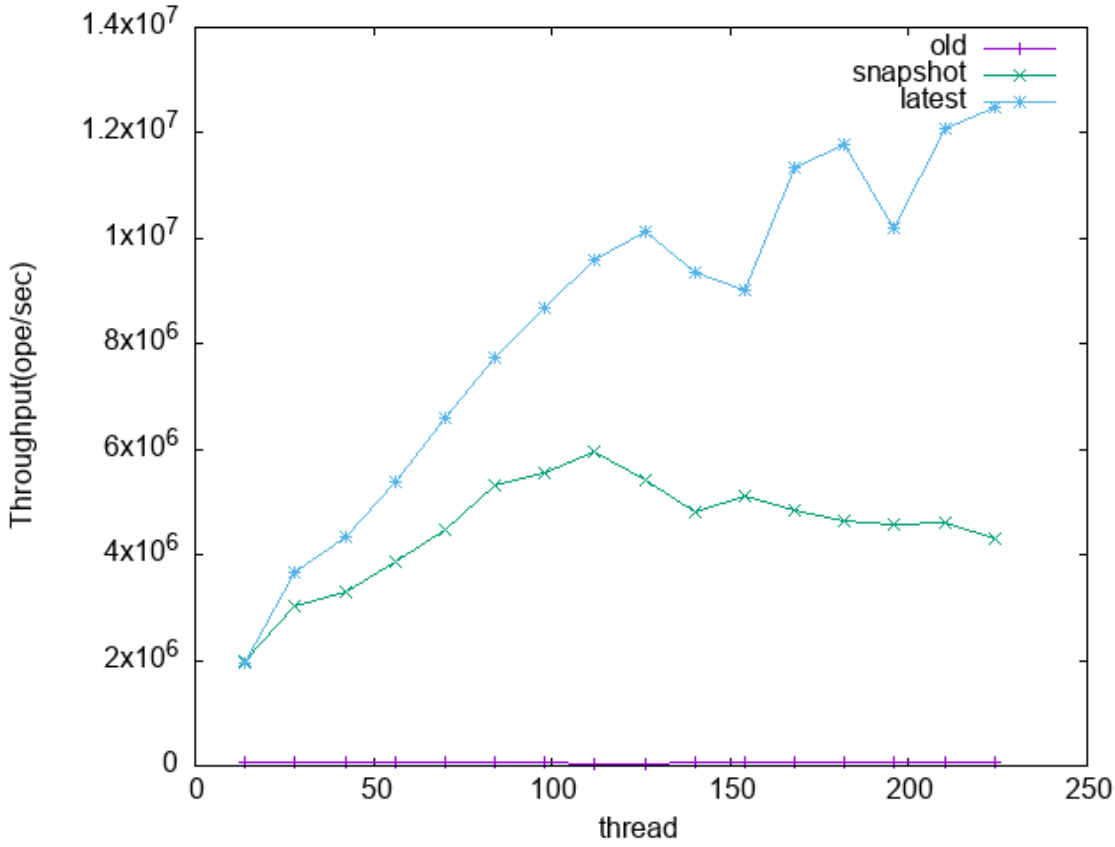

図 5.15: YCSB-B におけるスレッド数とスループットの関係

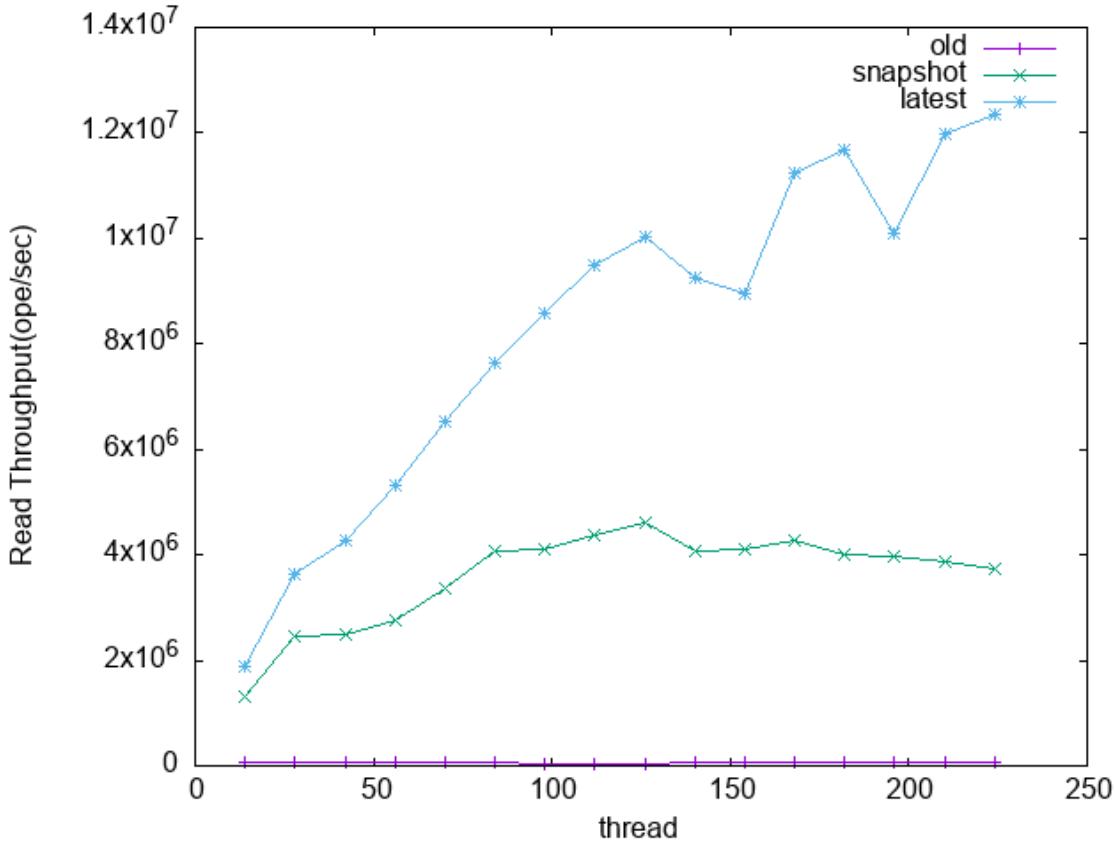

図 5.16: YCSB-B におけるスレッド数と読み込みスループットの関係

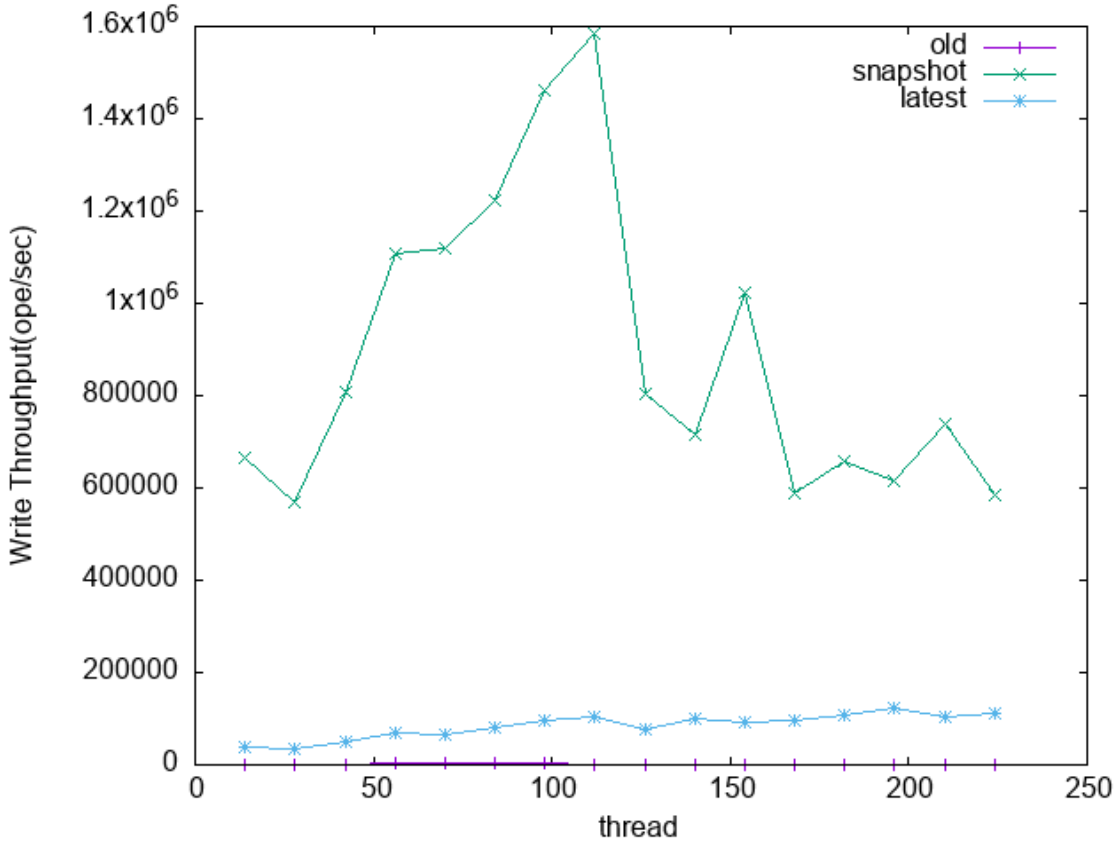

図 5.17: YCSB-B におけるスレッド数と書き込みスループットの関係

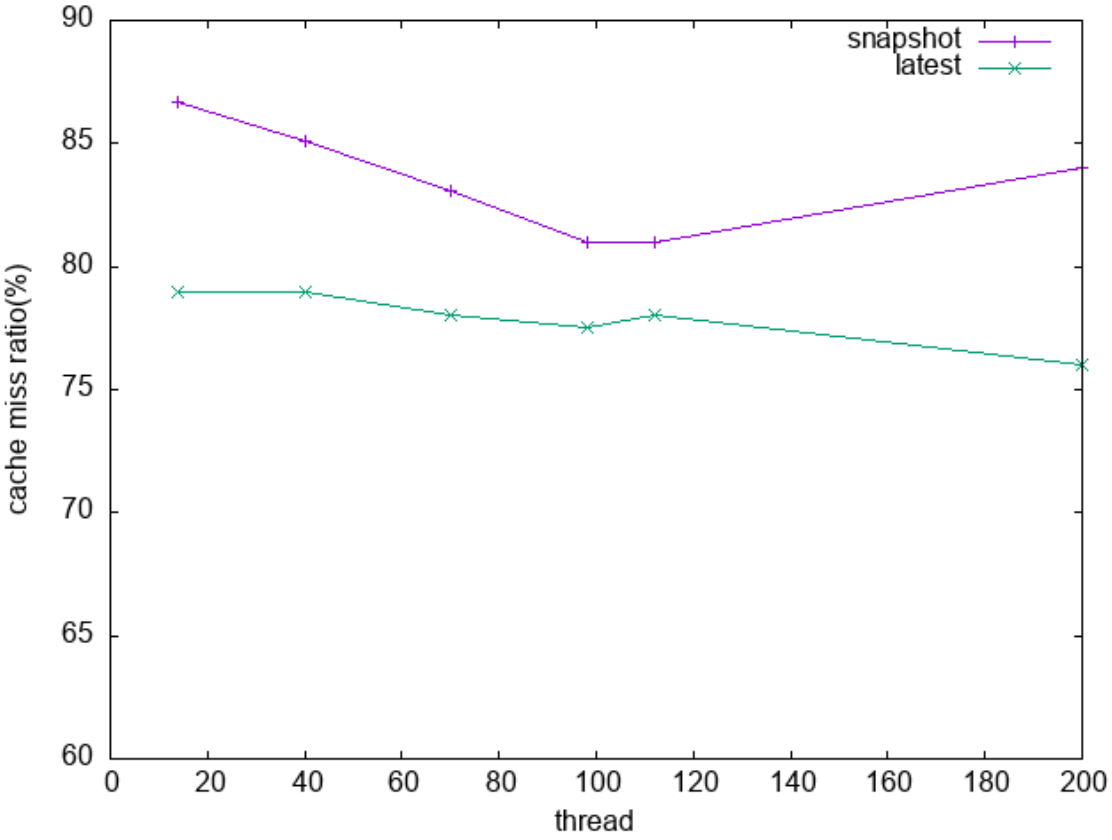

図 5.18: YCSB-B におけるスレッド数とキャッシュミス率の関係

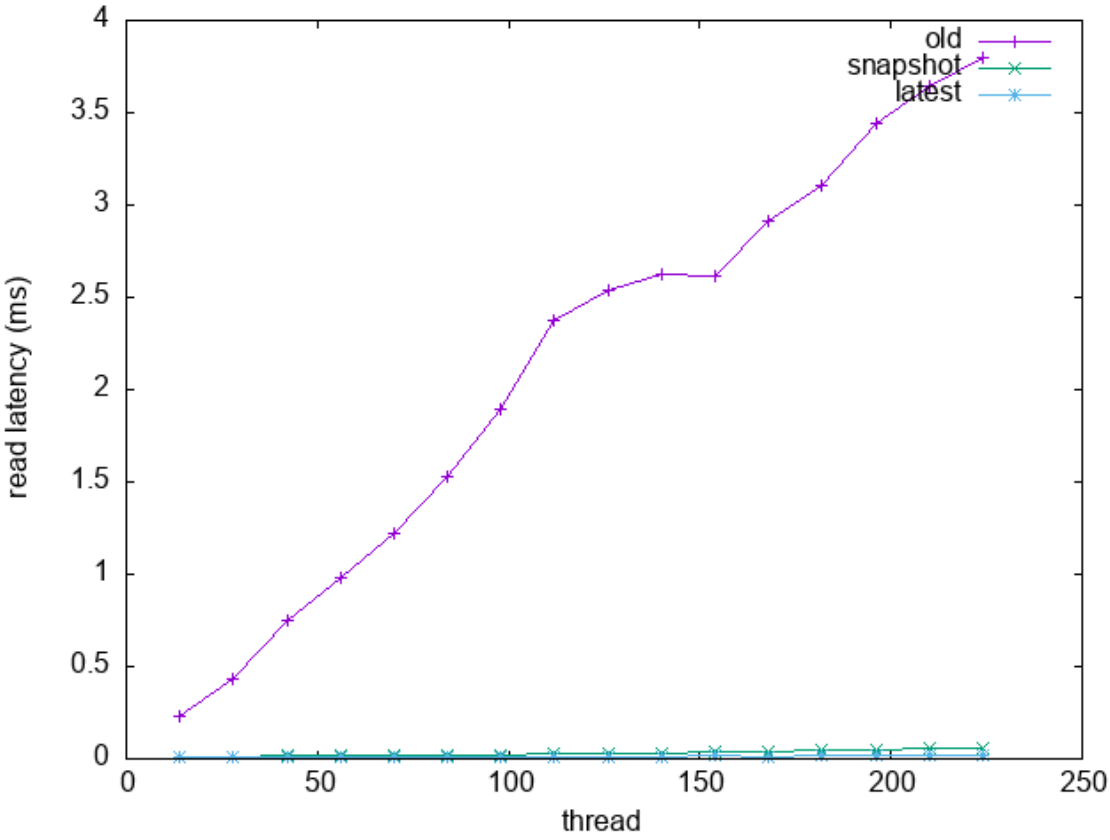

図 5.19: YCSB-B におけるスレッド数と読み込みレイテンシの関係

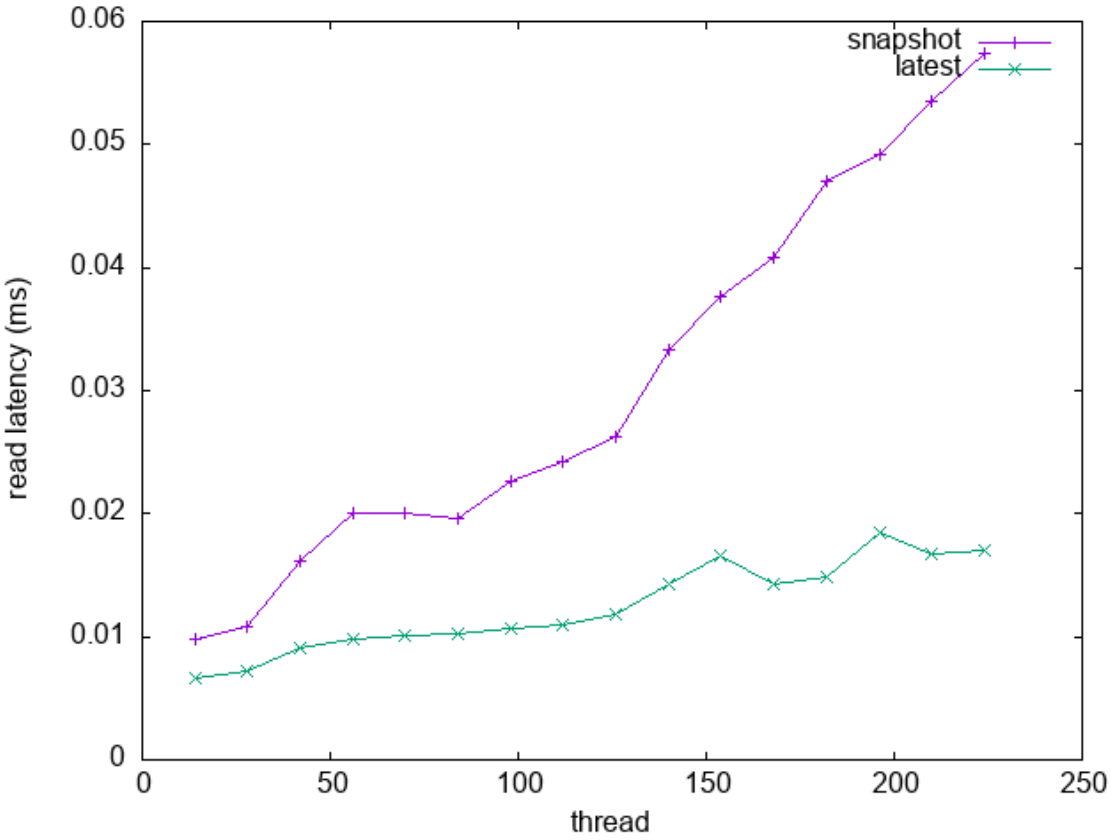

図 5.20: YCSB-B におけるスレッド数と読み込みレイテンシの関係 snapshot と latest のみ

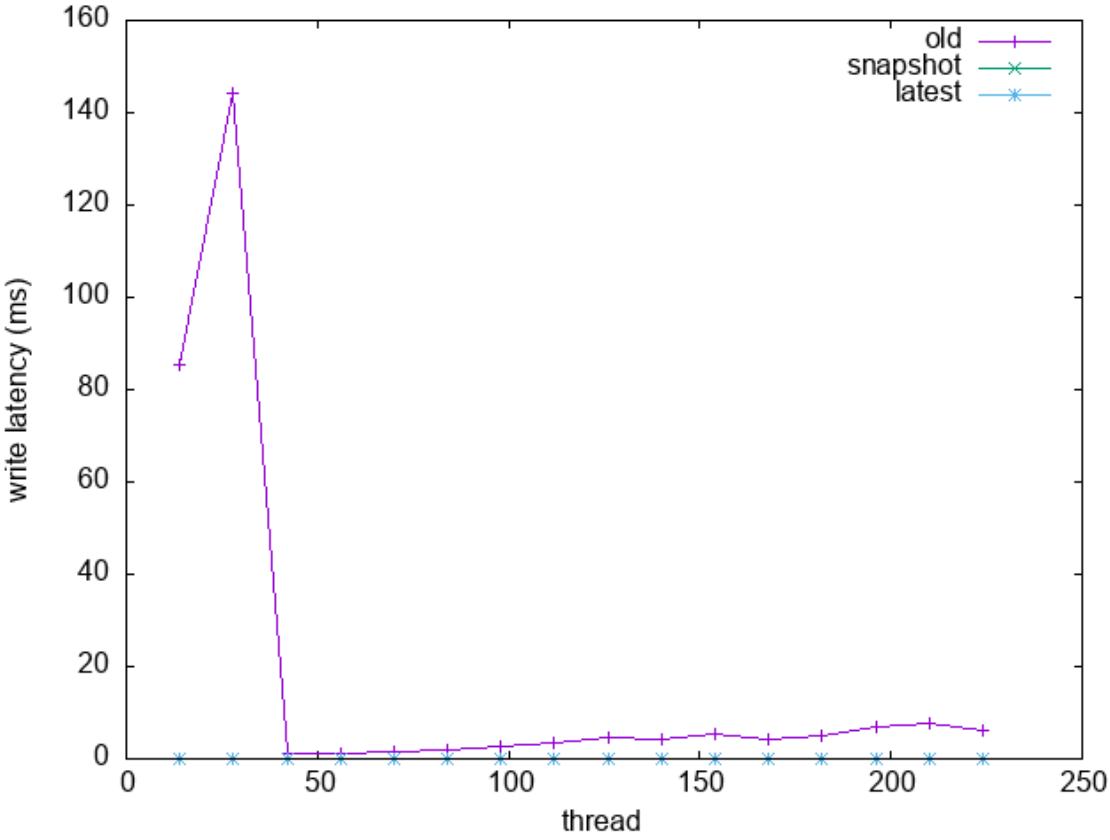

図 5.21: YCSB-B におけるスレッド数と書き込みレイテンシの関係

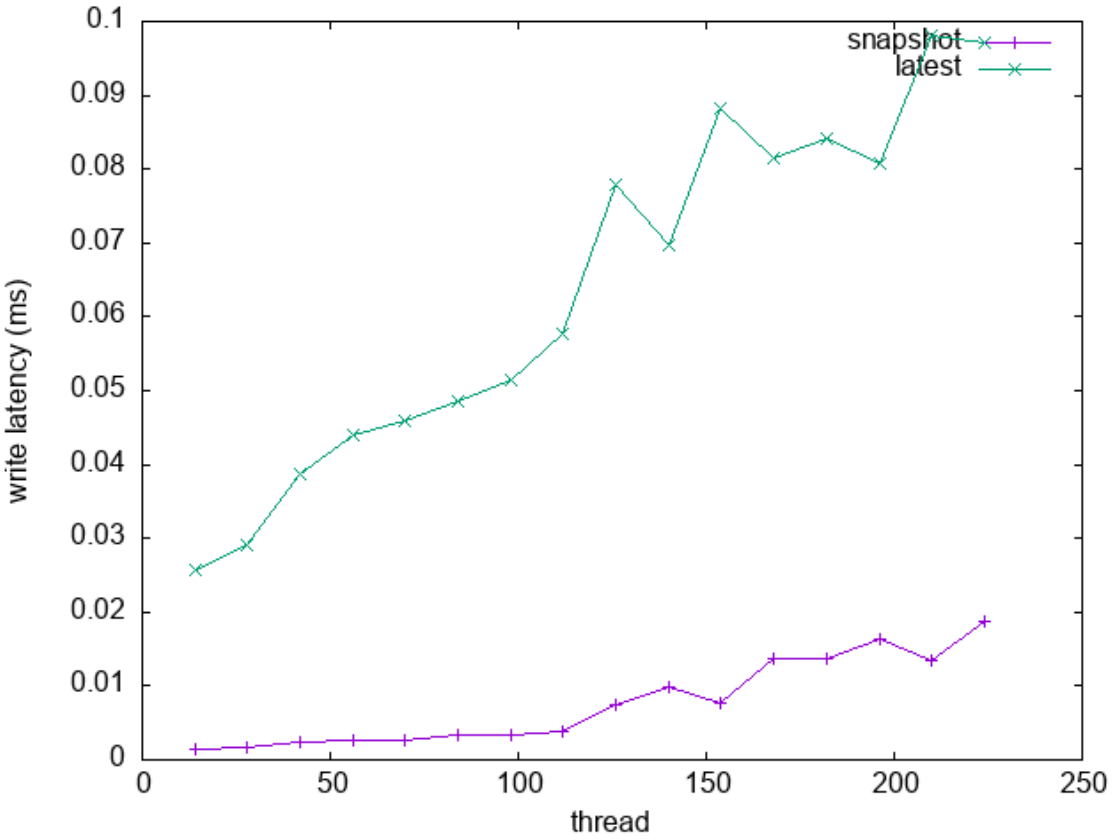

図 5.22: YCSB-B におけるスレッド数と書き込みレイテンシの関係 snapshot と latest のみ

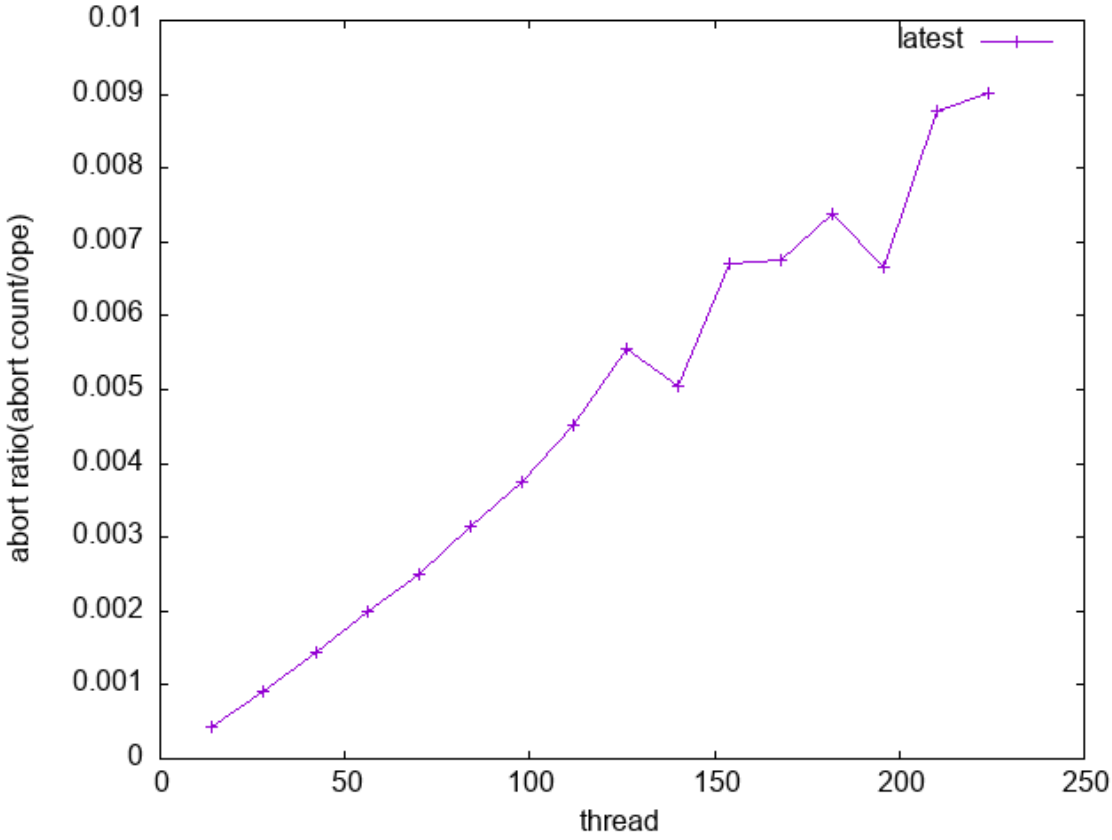

図 5.23: YCSB-B におけるスレッド数と abort 率の関係

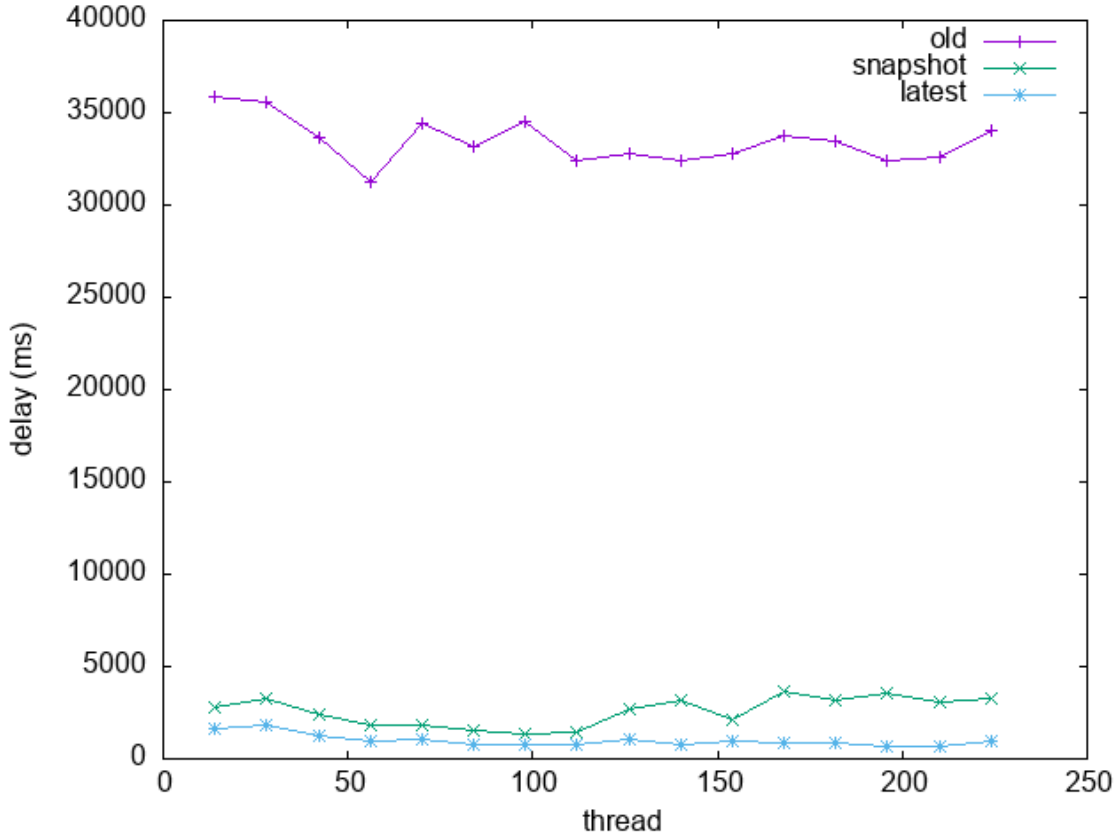

図 5.24: YCSB-B におけるスレッド数とデータの鮮度の関係

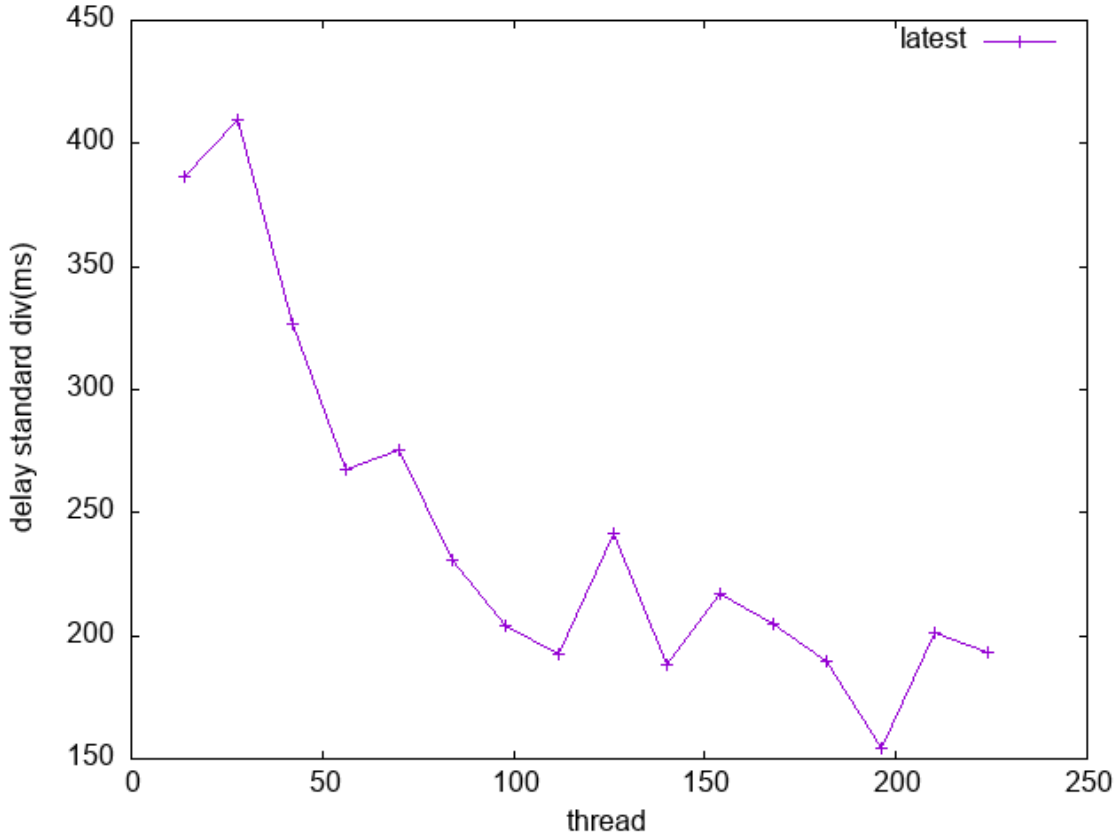

図 5.25: YCSB-B におけるスレッド数とデータの同期性の関係

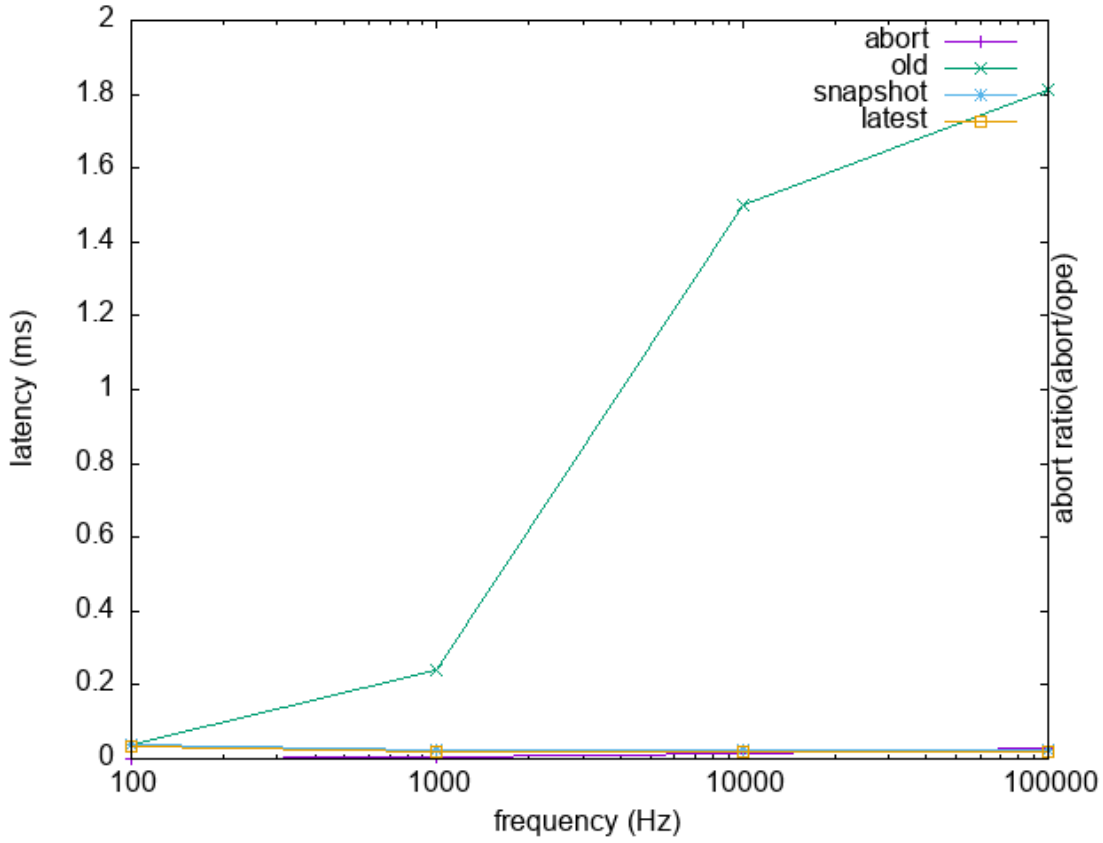

図 5.26: 制御周期とレイテンシ

第6章　結論と今後の課題

既存の TF ライブラリはジャイアンロックにより、アクセスが完全に逐次化され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があった。マルチコア化が進む現在のハードウェアの性能を活かせるよう、本研究では TF ライブラリに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用いてこの問題を解決した。その結果、既存手法と比べて最大 243 倍のスループット、最大 172 倍高速なレイテンシとなり、細粒度ロッキング法を導入した提案手法はマルチコアの性能を活かせ、優れた応答性能をもつ事が示せた。

また、既存の TF ライブラリはその仕様によって座標変換の計算時に最新のデータを参照せず、さらにジャイアントロックによりアクセスが逐次化されていることによってデータの鮮度が落ちるという問題があった。本研究ではデータベースの並行性制御技術における 2PL を適用することにより、複数の座標変換の最新のデータを atomic に取得できるインターフェイス (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イス (setTransforma) を提供することによって解決した。これにより、複数のデータの最新の座標変換情報の読み込み・書き込みを効率的に atomic に行えるようになった。その結果、既存手法と比べて最大 132 倍のデータ鮮度となり、2PL を導入した提案手法はデータの鮮度を向上させる事ができる事を示した。また、既存手法と比べ最大 257 倍のスループット、最大 282 倍高速化したレイテンシよな理、このインターフェイスはスループットとレイテンシにおいても既存手法より優れている事を示した。

本研究のようにロボットにおいてデータベースの並行性制御技術を導入する研究アプローチは他になく、ロボットの高性能化のためには並行性制御技術の導入が必要である事を示した。

本研究では TF ライブラリの森構造は変化しないと仮定したが、実際には時間とともにフレームが追加・削除され、またフレーム間の親子関係が変わることがある。データベースの並行性制御技術においてはこのような要素の追加・削除は IDM モデルで扱われ、高度な並行性制御には phantom anomaly を避ける必要がある。要素の追加・削除についても扱うことが今後の課題である。

本研究のアプリケーションは図3.3で取り上げたようなヘビ型ロボット以外にも、自動運転車におけるエッジクラウドでの応用が挙げられる。自動運転車におけるエッジクラウドでは、一部のローカルな区域における自動運転車両や発生した障害物の位置関係を管理することによって、渋滞の緩和や障害物の回避を行う事ができる。リアルタイム性が求められ、大量の自動運転車両や障害物の位置関係を TF ライブラリで管理するには、本研究で提案した手法を用いる事が有用である。また、以下のような問題にも対処する必要がある。

- 区域内への自動運転車両の出入りや障害物の登録・削除によって大量のフレームの追加・削除が発生するため、上述したような phantom anomaly の回避が必要になる
- TF ライブラリでは過去一定期間の座標変換情報を管理しているため、大量のフレームの追加・削除が発生する場合には適切な GC も必要になる

- エッジクラウドの電源がロストした場合にすぐに復旧し、いち早くサービスを提供できるように Crash Recovery を実装する必要もある

こういったアプリケーションにおいては、もはやデータベースの技術を応用することは避けられないだろう。

本研究では並行性制御の具体的なアルゴリズムとしては 2PL を実装したが、Silo を実装した場合の性能比較も行うことが今後の課題である。

また、よりロボット向けのワークロードに対応するため、TF に対する操作に優先順位をつけ、優先順位が高い操作をなるべく早く終わらせるために優先順位キューを実装することも検討が必要である。

ロボットに並行性制御技術を導入する対象として本研究では TF ライブラリを取り上げたが、他にも ROS で頻繁に使用される move_base などのパッケージにおいても並行性制御技術の導入の検討が必要である。

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] 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.

[2] 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.

[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.