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木構造として管理し、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。この有向木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法における 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

2022 年 01 月 14 日に REPORTOCEAN が発行した新しいレポートによると、自律移動型ロボットの世界市場は予測期間 2021-2027 年にかけて 19.6 %以上の健全な成長率で成長すると予測されている [1]。操縦者を必要とせず、自律的な移動が可能な自律移動型ロボットは、作業負担の大きい業務の代替や、労働力不足を解消する手段の一つとして関心が高まっている。小売・卸売倉庫ではインターネットショッピングによる受発注作業の継続的な増加と人手不足が深刻化している。企業は、こうした課題解決のために、商品棚やパレットを運ぶ自律移動型ロボットの導入を加速させており、さらに商品管理から出荷前の棚出し梱包までを自動化することで、作業員の作業量削減と能力に依存しないオペレーションの構築を目指している。このような自動化の実現に向けて、自律移動型ロボットが積極的に活用されていくとされる [2]。

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 [3]。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。直角に交わる二つの矢印は座標系を表し、交点が座標系の原点、二つの矢印が反時計回りにそれぞれ X・Y 軸を表す。

座標系とは空間中の物体の座標を定める方法を与えるものであり、座標系内の物体は原点からの X・Y 軸方向の距離の組で座標が与えられる。空間全体の座標系をグローバル座標系と呼び、その中にある個別の物体それぞれにローカル座標系を設定することによって、全体空間の中でのそれぞれのオブジェクトの変化を扱いやすくする。ここでは地図、ロボット、オブジェクトそれぞれに座標系が与えられ、グローバル座標系として地図座標系、ローカル座標系としてロボット座標系・オブジェクト座標系として扱う。地図座標系の原点は地図の左下端に設定され、ロボット座標系・オブジェクト座標系はそれぞれロボット・オブジェクトの中心点と向きから決められる。

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し、この位置関係は三次元ベクトルで表現される平行移動成分と、四元数で表現される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 [3]。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。これを表1.1にまとめた。

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

座標変換	座標変換を計算するプログラム	座標変換に使われるデータ
地図座標系からロボット座標系	自己位置推定プログラム	LiDAR からの点群データ
ロボット座標系から物体座標系	物体認識プログラム	カメラからの画像データ

表 1.1: 座標系間の位置関係を更新するセンサーとプログラム

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のそれぞれが登録されたタイミングを黒いセルで表し、それぞれが異なるタイミングで登録されることを示している。

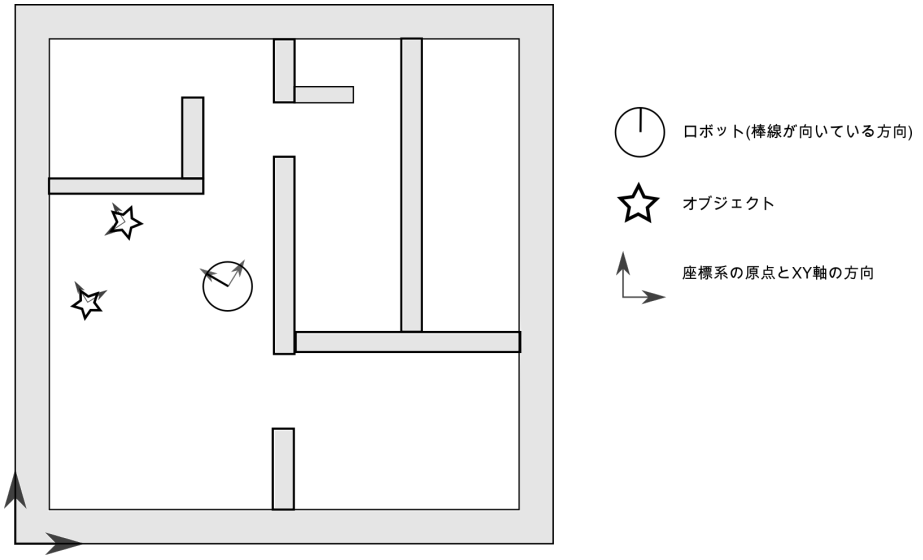

図 1.1: 部屋の中のロボット

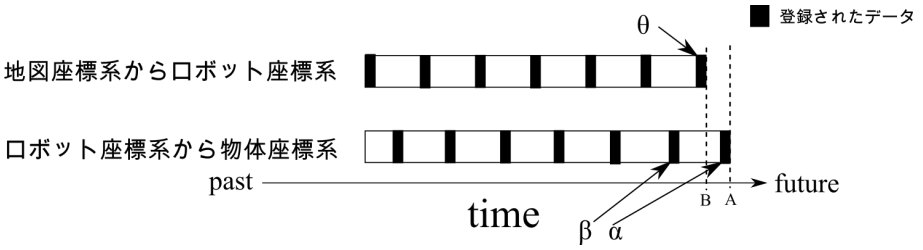

図 1.2: 位置関係の登録のタイムライン

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。三次元ベクトル $(x, y, z)^{\mathrm{T}}$ を同次座標で表した $\mathbf{p} = (x, y, z, 1)^{\mathrm{T}}$ と、四元数 $\mathbf{q}$ で表現される回転がある時、$\mathbf{p}$ を $\mathbf{q}$ で回転して得られる同次座標 $\mathbf{p}'$ は

$$\mathbf{p}' = \mathbf{q}\mathbf{p}(\mathbf{q})^{-1} \tag{1.1}$$

で与えられる。これにより、地図座標系からロボット座標系への位置関係が平行移動成分が $\mathbf{p}_{\mathrm{map}\to\mathrm{robot}}$、回転成分が $\mathbf{q}_{\mathrm{map}\to\mathrm{robot}}$、ロボット座標系から物体座標系への位置関係が平行移動成分が $\mathbf{p}_{\mathrm{robot}\to\mathrm{object}}$、回転成分が $\mathbf{q}_{\mathrm{robot}\to\mathrm{object}}$ で与えられる時、地図座標系から物体座標系への平行移動成分 $\mathbf{p}_{\mathrm{map}\to\mathrm{object}}$、回転成分 $\mathbf{q}_{\mathrm{map}\to\mathrm{object}}$ はそれぞれ

$$\mathbf{p}_{\text{map}\to\text{object}} = \mathbf{q}_{\text{robot}\to\text{object}}\mathbf{p}_{\text{map}\to\text{robot}}(\mathbf{q}_{\text{robot}\to\text{object}})^{-1} \tag{1.2}$$
$$\mathbf{q}_{\text{map}\to\text{object}} = \mathbf{q}_{\text{robot}\to\text{object}}\mathbf{q}_{\text{map}\to\text{robot}}$$

で与えられる。このように、二つの座標系間の位置関係はその座標系どうしを繋ぐような座標変換を平行移動成分、回転成分それぞれ掛け合わせることによって計算できる。

ロボット座標系から地図座標系への位置関係(平行移動成分 $\mathbf{p}_{\text{robot}\to\text{map}}$ と回転成分 $\mathbf{q}_{\text{robot}\to\text{map}}$)を得るには、地図座標系からロボット座標系への位置関係の逆変換を取れば良い。これは、

$$\mathbf{q}_{\text{robot}\to\text{map}} = \mathbf{q}_{\text{map}\to\text{robot}}{}^{-1} \tag{1.3}$$
$$\mathbf{p}_{\text{robot}\to\text{map}} = \mathbf{q}_{\text{robot}\to\text{map}}(-\mathbf{p}_{\text{map}\to\text{robot}})(\mathbf{q}_{\text{robot}\to\text{map}})^{-1}$$

で与えられる。

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次のように複雑なものとなる。

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 α 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時刻 A にはまだ存在しない。このため、最新の変換データ θ を使う、もしくは過去のデータを元にデータの補外をし、時刻 A 時点での変換データを予測するという選択肢が挙げられる。

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変換データ θ 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 β か α のデータを利用するという選択肢が挙げられる。

各変換データが異なるタイミングで来るために選択肢が複数生じるという問題以外にも、情報の分散という問題もある。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表1.1のように別々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別々のプログラムで管理され、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 [3]。

1.2 TF ライブラリ

この問題を解決するために TF ライブラリが提案された。TF ライブラリは ROS 上で動作し、各座標系間の変換を有向木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 [3]。図1.1を表す木構造は図1.3で表現できる。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変換データが存在することを表す。

ノードは TF ではフレームと呼ばれ、フレーム中の文字列は各座標系に対応するフレーム名である。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。親フレームへ張られたエッジは子フレームから親フレームへポインタが貼られていることを表し、子フレームから親フレームを辿ることができる。しかしながら、親フレームから子フレームを辿ることはできない。このた

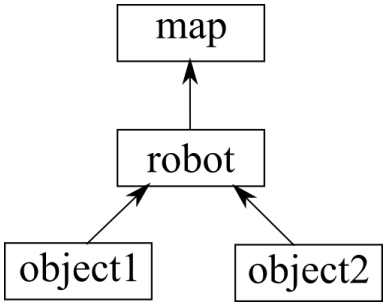

図 1.3: 図1.1に対応する木構造

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。

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 10 秒間保存する。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 はある時刻の座標変換データが保存されているか線形補間で取得できる時に、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。

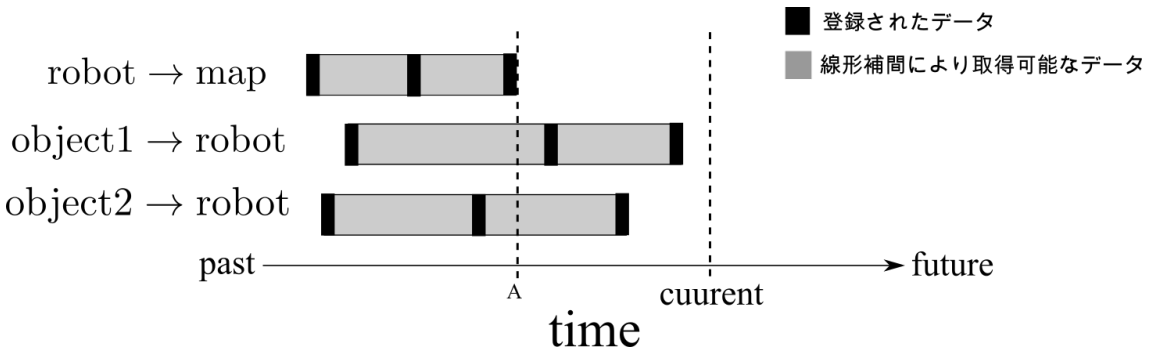

図 1.4: 図1.3における位置関係登録のタイムライン

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 への各エッジを確認する。ここでは object1 から map への各エッジは object1→robot, robot→map であることがわかる。

次に、どのエッジ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 での各エッジのデータを取得し、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。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の木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木構造にアクセスしている際は他のスレッドは木構造にアクセスできないアルゴリズムとなっている。複数スレッドが木構造の別々の部分にアクセスするケース、及び複数スレッドが木構造の同じ部分のアクセスしているが全て読み込みアクセスのケースなど、排他制御が必要ではないケースにおいてもアクセスが完全に逐次化されている。これにより、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問題が生じる可能性がある。

問題2a：データの鮮度

上述のように、TFのフレーム間の座標変換計算インターフェースはある時刻の座標変換データが保存されているか線形補間で取得できる時に、その時刻の座標変換データを提供できるという仕様のため、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時刻同期性はあるが、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。これにより、ロボットの制御や自己位置推定に問題が生じる可能性がある。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。

また、この仕様により座標系間の位置関係があまり変わらない場合についても頻繁にデータを登録する必要があり、無駄な処理が発生する。

問題2b：データの一貫性

問題2aの解決策として、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を提供するだけでは不十分である。これは、複数の座標変換データを登録している途中に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と、ユーザーが期待するデータの一貫性がなくなる可能性があるからである。

1.4 貢献

問題1については、並行性制御法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フレーム単位でのロックとなり、複数のフレームに並行にアクセスが可能になった。細粒度ロックを実装した場合のスループットは最大で11,574,200tps、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。

問題 2a、2b については、データベースの並行性制御法における 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 (lookupLatestTransformXact)、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 (setTransformsXact) 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、lookupLatestTransformXact と setTransformsXact を使うことによりこれは解決できた。既存手法と比べ、lookupLatestTransformXact と setTransformsXact 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。既存の TF ライブラリ [21] に 1236 行分の変更を加え、提案手法を実装したコードを [23] に公開した。

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 の木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木構造への再粒度ロックの導入とデータの鮮度、データの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

2.1　データベース分野におけるロボット研究

筆者の知る限り、データベース分野ではロボットを対象にした研究論文はトップ会議、トップジャーナルでは発表されたことがない。他方、産業界における珍しい例としては GAIA platform [15] 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。このデータベースアクセスはトランザクションを用いて実行される。GAIA チームには snapshot isolation 提案者も含まれており、そのトランザクションアーキテクチャには一定の頑健性があることが期待される。

2.2　ロボット分野におけるデータベース研究

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 [5] 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 [16] 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。

2.3　トランザクション研究

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 [6]、Silo [8]、MOCC [7]、Cicada [9] が挙げられる。古典的な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劇的に高い性能を実現する。具体的な理由は contetion point の消失である。Multi-version concurrency control (MVCC) の場合、single shared counter を用いるため、メニーコア環境では性能が劇的に劣化する。そのため、snapshot isolation や multi-version timestamp ordering のような scheduling space の広大な手法であっても、高い性能を出すことは難しい。メニーコア環境では MVCC よりも 2PL の方が高い性能を有することが知られている [10].

第3章　既存のTFの木構造とその問題点

3.1　TFライブラリの使用例

図 3.1: ロボット「Aqua」

ロボット「Aqua」は筑波大学知能ロボット研究室で開発された自律移動ロボットであり、前輪 2 輪が駆動輪、後輪のキャスタによる従動輪で構成される差動 2 輪駆動方式のロボットである [11]。制御には T-frog Project のモータドライバ TF-2MD3-R6 を使用し、本ロボットの移動制御を行う。計算機部分については CPU は Intel Core i7-9700K Processor、GPU は GeForce GTX 1050 Ti、RAM は 32GB で構成されている。機体の上部、旋回中心に自己位置推定に使用する Velodyne 社製の VLP-16 が 1 台、前後方の障害物の検出に使用する北陽電

機社製の UTM–30LX を機体下部前方と後方に 2 台搭載している。また、BUFFALO 社製の Web カメラ BSK200MBK が 3 台、PointGray 社製の Grasshopper3 カメラ GS3-U3-123S6M-C を用いて構築したステレオカメラと北陽電機社製 YVT–X002 をそれぞれ 1 台搭載している。Aqua の外観を図3.1に示す。

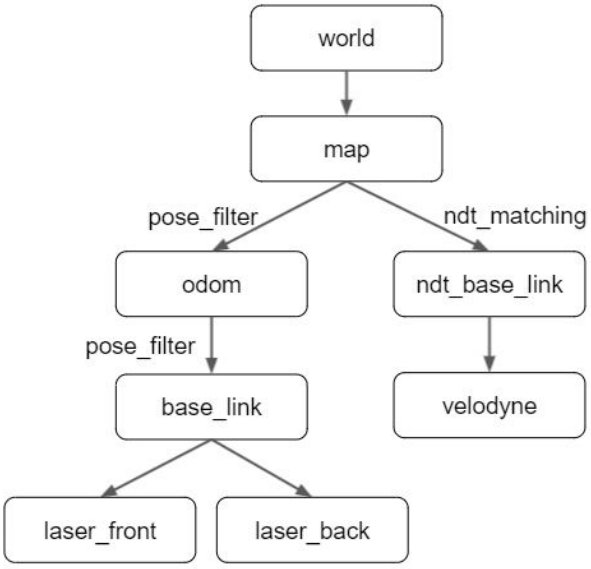

図 3.2: Aqua における TF の木構造、[11] より引用。

Aqua のシステムではロボットの自己位置とセンサーの位置を管理するために TF ライブラリを用いている。Aqua における TF の木構造を図3.2 [11] に示す。エッジの横に書かれているのはフレーム間の座標変換を更新するプログラム名である。 Aqua では、VLP-16 からの三次元点群データを元に、Autoware [17] の ndt_matching モジュール [18] を用いて自己位置推定を行う。ndt_matching によって与えられる自己位置推定は三次元空間での自己位置のため、Aqua ではその三次元空間での自己位置を二次元空間での自己位置に変換する pose_filter パッケージを自作して、ロボットの自己位置を TF ライブラリに登録する。

このように、TF ライブラリは自律移動ロボットでは広く利用されている。

3.2 構造

TF ライブラリでは図3.3のように各座標系間の位置関係を木構造で管理する。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。ノードはフレームと呼ばれ、フレーム内の文字列は各座標系に対応するフレーム名である。

各フレーム間の座標変換情報は、フレーム間のエッジに 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3.4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。

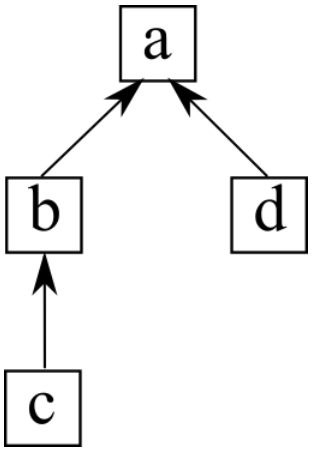

図 3.3: TF の木構造

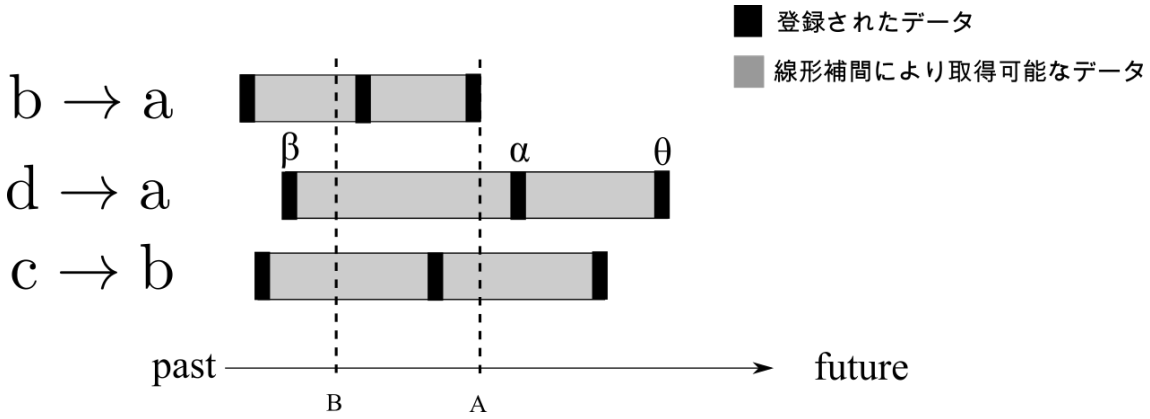

図 3.4: タイムライン

3.3 TF の木構造のインターフェイス

TF ライブラリの木構造のインターフェイスは [13] に公開されており、このうち主に以下の表3.1のインターフェイスが使われている。

この中でも頻繁に使用されるのが *lookupTransform* と *setTransform* であり、TF の木構造へのアクセスは主に、高頻度で *lookupTransform* のみ呼び出す読み込み専用スレッドと、高頻度で *setTransform* のみ呼び出す書き込み専用スレッドで構成される。本研究ではこの二つのインターフェイスの改善を行う。

3.4 lookupTransform

二つのフレーム間の座標変換情報を取得するには *lookupTransform* を使う。lookupTransform の実装は [14] に公開されており、登録されたフレームが図3.3、登録された座標変換情報のタイムラインが図3.4の状況において、*lookupTransform* を用いてフレーム c からフレーム d への座標変換を計算する動作例を説明する。

1. フレーム c から木構造のルートフレームへのエッジを取得する。ここではルートフレームは a であり、フレーム c からフレーム a へのエッジは c→b と b→a である。

setTransform	フレーム間の座標変換を登録する
lookupTransform	フレーム間の座標変換を計算する
canTransform	フレーム間の座標変換が計算できるかを確認する
allFramesAsYAML	全てのフレームの情報を YAML 形式で取得する
allFramesAsString	全てのフレームの情報を文字列で取得する
addTransformableCallback	フレーム間の座標変換が登録された時のコールバック関数を追加する
removeTransformableCallback	addTransformableCallback で追加したコールバック関数を削除する

表 3.1: TF の木構造のインターフェイス

2. フレーム d から木構造のルートフレームへのエッジを取得する。同じようにルートフレームは a となり、フレーム d からフレーム a へのエッジは d→a となる。

3. 得られた三つのエッジである c→b、b→a、d→a から、どのエッジ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 における各エッジ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でデータを生成する。

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* 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フレームへの時刻 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フレームへのエッジは 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フレームへの時刻 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フレームへのエッジは 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 の疑似コードをアルゴリズム 1で説明する。*lookupTransform* にて最新の座標変換を計算するには3行目のように、まず *getLatestCommonTime* インターフェイスを呼び出し、source フレームから target フレームへの最新の座標変換を計算できる時刻を取得する。

source フレームと target フレームの位置関係には以下の四パターンがあり、それぞれ図3.5〜図3.8で示す。図中の s は source フレームを、t は target フレームを表す。

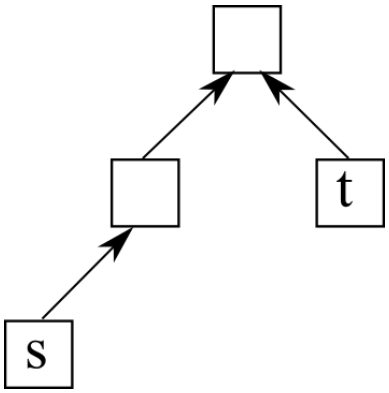

図 3.5: source と target の祖先が root のみのパターン

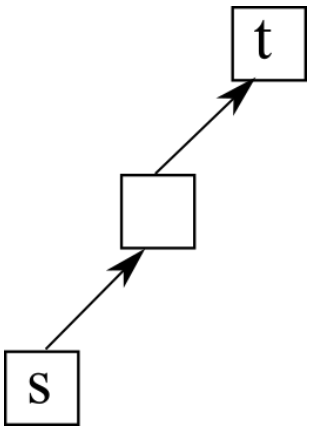

図 3.6: source の祖先が target のパターン

1. source と target の祖先が root のみのパターン
2. source の祖先が target のパターン
3. target の祖先が source のパターン
4. source と target の祖先が root だけではないパターン

このうち、2〜4 のパターンについては source フレームから target フレームへの最新の座標変換を計算できる時刻を取得する際に、次に説明するように source から root へ、もしくは target から root へ辿る処理を省略できる。

パターン 2 では source から root へのパスを辿る際に途中で target を見つけ、root まで辿ること無しにその時点で取得できたデータから最新の座標変換を計算できる時刻を取得できる。同じように、パターン 3 では target から root へのパスを辿る際に途中で source を見つけ、その時点で取得できたデータから最新の座標変換を計算できる時刻を取得できる。

1 と 4 との違いについて説明する。図3.8はパターン 4 の状況を表し、p が source と target 共通の祖先のフレーム、r が root フレーム、t_1、t_2、t_3 がそれぞれ p→r、s→p、t→p の座標変換を取得できる最新の時刻を表す。source フレームから target フレームへの最新の座標変換を計算できる時刻は $\min(t_2, t_3)$ なので t_1 は無視できる。このため、source から root へ辿る時にアクセスしたフレームを記録しておき、次に target から root へ辿る時に記録を元にして root でない共通の祖先を見つけられれば、target から root へ辿る処理を省略できる。

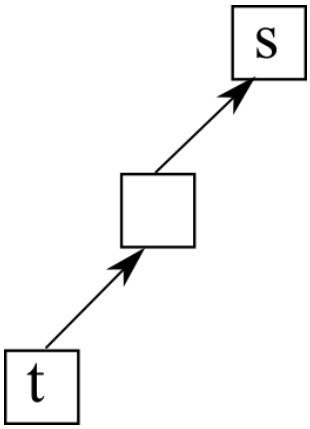

図 3.7: target の祖先が source のパターン

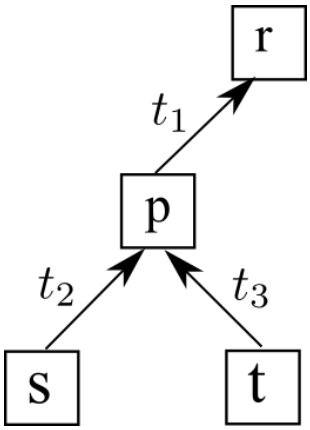

図 3.8: source と target の祖先が root だけではないパターン

getLatestCommonTime ではまず、source フレームからルートフレームへのエッジをたどり、各エッジで最新の座標変換を提供できる時刻の最小値を取る (5〜13行目)。これにより、source フレームからルートフレームへの最新の座標変換を取得できる時刻を取得できる。source フレームからルートフレームへのエッジをたどるときに各エッジの情報を lct_cache 変数に記録しておく (8行目)。10行目は上述のパターン 2 の、target が source の祖先である場合の処理を示している。

続いて、target フレームからルートフレームへのエッジをたどり、同じようにして target フレームからルートフレームへの最新の座標変換を取得できる時刻を取得する (16〜27行目)。19行目はパターン 4 の、source と target が root 以外にも同じ祖先を持つ場合の処理を示し、target フレームからルートフレームへ辿る処理を省略している。24行目は上述のパターン 3 の、target が source の祖先である場合の処理を示している。

最後に、source フレームからルートフレームへの最新の座標変換を取得できる時刻、target フレームからルートフレームへの最新の座標変換を取得できる時刻から、source フレームから target フレームへの最新の座標変換を取得できる時刻を取得する (28 〜33行目)。上述のパターン 4 に対応するために、lct_cache 変数内の記録から最新の座標変換を取得できる時刻を取得している。

source フレームから target フレームへの最新の座標変換を計算できる時刻を取得できたため、処理は *lookupTransform* に戻り、取得できた時刻は time 変数に代入される。

lookupTransform においても、*getLatestCommonTime* と同じように各パターンへの最適化ができる。上述のパターン 2 と 3 では root まで辿ること無しにその時点で取得できた座標変換情報を利用できる。パターン 4 においては最適化は行わず、パターン 1 と同じように source フレームからルートフレームへの座標変換と、ルートフレームから target フレームへの座標変換を掛け合わせる。

まず、source フレームからルートフレームへのエッジをたどり、source フレームからルートフレームへの座標変換を計算する (8〜16行目)。21行目は上述のパターン 2 の、target が source の祖先である場合の処理を示している。

続いて、target フレームからルートフレームへのエッジをたどり、target フレームからルートフレームへの座標変換を計算する (19〜26行目)。10行目は上述のパターン 3 の、source が target の祖先である場合の処理を示している。

最後に、source フレームからルートフレームへの座標変換にルートフレームから target フレームへの座標変換を掛け合わせることにより、source フレームから target フレームへの座標変換を取得できる。ルートフレームから target フレームへの座標変換は、target フレームからルートフレームへの座標変換の逆変換を取ることにより取得できる。

lookupTransform で最新の座標変換情報を得る場合には内部で *getLatestCommonTime* を呼び出しているため、同じフレームに 2 度アクセスすることになる。

3.5 setTransform

二つのフレーム間の座標変換情報を更新するには *setTransform* を使う。図3.3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。

このインターフェイス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。座標変換情報は 10 秒間保存される。

setTransform の擬似コードをアルゴリズム3に示す。

3.6 問題点

問題 1: ジャイアント・ロック

上述のように、TF ライブラリの木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木構造全体を保護している。このため、一つのスレッドが木構造にアクセスしている際は他のスレッドは木構造にアクセスできないアルゴリズムとなっている。複数スレッドが木構造の別々の部分にアクセスするケース、及び複数スレッドが木構造の同じ部分のアクセスしているが全て読み込みアクセスのケースなど、排他制御が必要ではないケースにおいてもアクセスが逐次化されおり、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問題が生じる可能性がある。

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。ヘビ型ロボットとは生物のヘビを模倣したロボットであり、ヘビのように関節を動かすことによ

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  ▷ source から root まで辿る
9:         (trans, parent) = frame.getTransAndParent(time)
10:        if frame == target then  ▷ target が source の祖先の場合
11:            return source_trans
12:        end if
13:        source_trans *= trans  ▷ 座標変換の掛け合わせ
14:        top_parent = frame
15:        frame = parent
16:    end while
17:    frame = target
18:    target_trans = I
19:    while frame ≠ top_parent do  ▷ target から root まで辿る
20:        (tarns, parent) = frame.getTransAndParent(time)
21:        if frame == source then  ▷ source が target の祖先の場合
22:            return (target_trans)^-1
23:        end if
24:        target_trans *= trans
25:        frame = parent
26:    end while
27:    return source_trans * (target_trans)^-1  ▷ source から root への変換 * root から source への変換
28: 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 = [ ]                                  ▷ source から root へ辿るときの履歴
5:     while frame ≠ root do                            ▷ source から root まで辿る
6:         (time, parent) = frame.getLatestTimeAndParent()
7:         common_time = min(time, common_time)
8:         lct_cache.push_back((time, parent))
9:         frame = parent
10:        if frame == target then                      ▷ target が source の祖先の時
11:            return common_time
12:        end if
13:    end while
14:    frame = target
15:    common_time = TIME_MAX
16:    while true do                                    ▷ target から root まで辿る
17:        (time, parent) = frame.getLatestTimeAndParent()
18:        common_time = min(time, common_time)
19:        if parent in lct_cache then                  ▷ source と target が同じ祖先を持つとき
20:            common_parent = parent
21:            break
22:        end if
23:        frame = parent
24:        if frame == source then                      ▷ source が target の祖先の時
25:            return common_time
26:        end if
27:    end while
28:    for (time, parent) in lct_cache do               ▷ lookup tree cache を元に、最新の時刻を取得する
29:        common_time = min(common_time, time)
30:        if parent == common_parent then
31:            break
32:        end if
33:    end for
34:    return common_time
35: 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
```

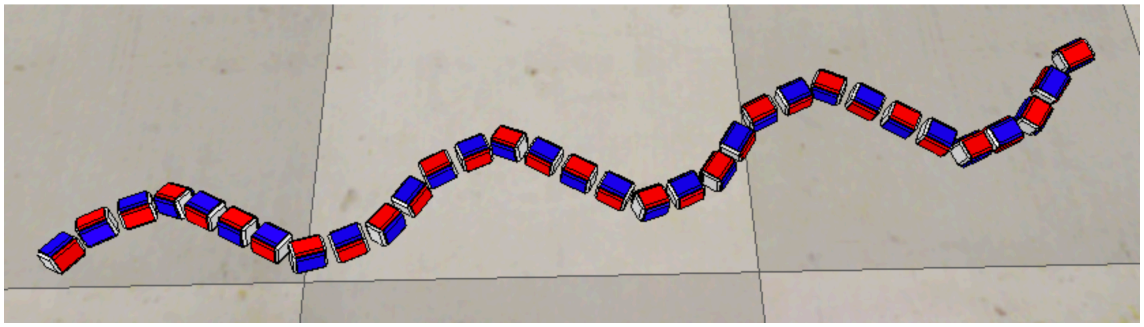

図 3.9: シミュレータ上で動作するヘビ型ロボット、[12] より引用。

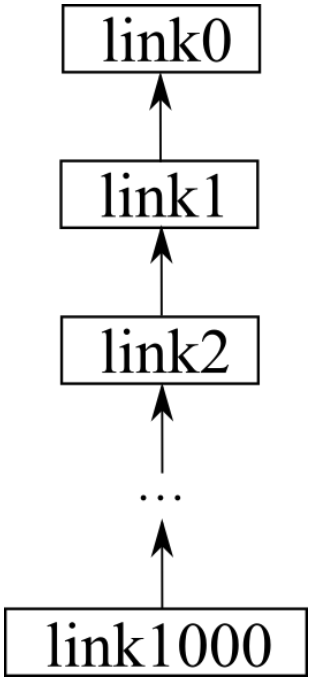

図 3.10: ヘビ型ロボットにおける木構造

り狭い場所の点検や災害現場での被災者探索に用いられている。図3.9はシミュレータ上で動作する関節数が 30 のヘビ型ロボットを示す [12]。

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3.10のように比較的巨大な木構造になる。§ 3.3にて述べたように、TF の木構造へのアクセスパターンは主に木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、木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る状況なので、それぞれのスレッドが木構造の別の部分にアクセスするにもかかわらず、一つのスレッドが木構造にアクセスするたびに木構造全体がロックされてしまいパフォーマンスに問題が生じる可能性がある。

問題 2a: データの鮮度

木構造が図3.3、タイムラインが図3.4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、フレーム間のエッジ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。同時刻のデータを元に座標変換を計算する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。これにより、ロボットの制御や自己位置推定に問題が生じる可能性がある。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。また、次に説明するようにこの仕様によって座標系間の位置関係があまり変わらない場合についても頻繁にデータを登録する必要があり、無駄な処理が発生する。

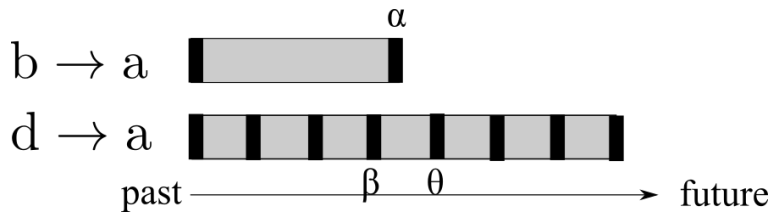

図 3.11: 不必要な更新が必要な例

図3.11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「ある時刻の座標変換データが保存されているか線形補間で取得できる時に、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る。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る。

問題 2b: データの一貫性

問題 2a の解決策として、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を提供するだけでは不十分である。これは、複数の座標変換データを登録している途中に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と、ユーザーが期待するデータの一貫性がなくなる可能性があるからである。

第4章　提案手法

本研究では、データベースの並行性制御法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

上述した問題 1 については、データベースの並行性制御法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の木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。

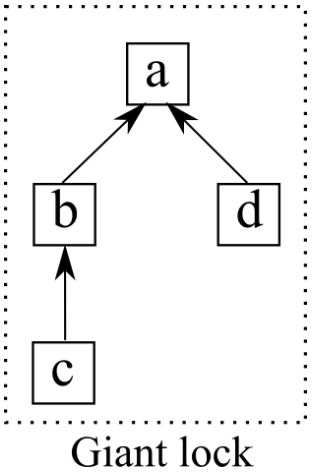

図 4.1: Giant lock

TF ライブラリでは木構造へのアクセスをする際、木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 は木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 が木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 の処理が完了し木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。

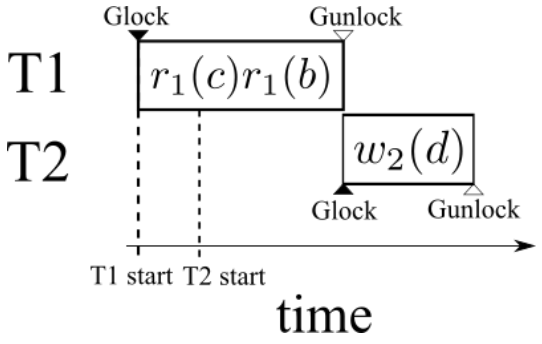

図 4.2: ジャイアントロックにおけるスケジュール

そこで、本研究ではデータベースの並行性制御法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確保し、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、ロックを確保する。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確保する必要がある
- 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確保できる
- 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確保することができる
- 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確保することはできない
- 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確保することはできない

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

表 4.1: 各ロックの互換性

このルールを表4.1にまとめる。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 によってロックが確保された状態を表し、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 $j(i \neq j)$ が確保できるかどうかを 2、3 行目で表している。∘ はすでにロックが確保されていても別のスレッドがロックを確保できることを表し、× はそうでないことを表す。例えば、2 行 2 列目はすでに $rl_i(x)$ が確保されていても $rl_j(x)$ が確保できることを表し、2 行 3 列目はすでに $wl_i(x)$ が確保されていると $rl_j(x)$ は確保できないことを表す。

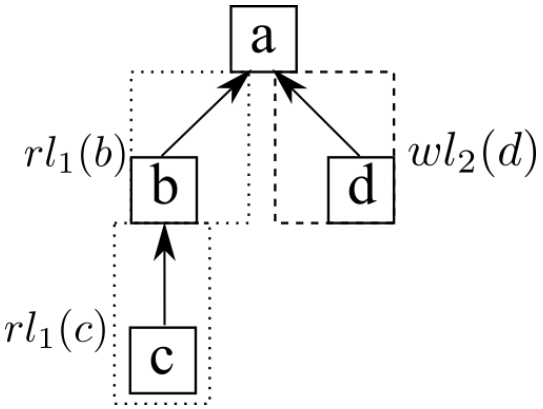

図 4.3: 細粒度ロッキング

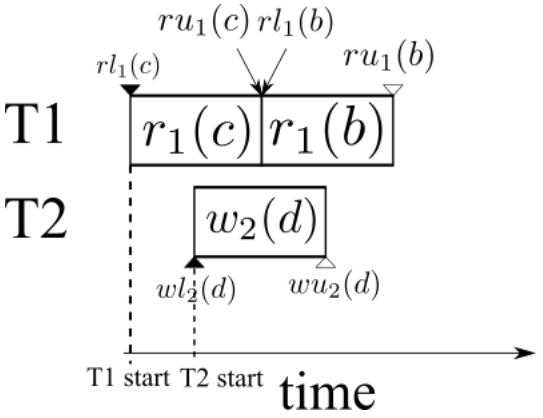

図 4.4: 細粒度ロックにおけるスケジュール

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

$$rl_1(c)r_1(c)wl_2(d)w_2(d)ru_1(c)rl_1(b)r_1(b)wu_2(d)ru_1(b) \tag{4.1}$$

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確保す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確保さ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

$$rl_1(c)r_1(c)rl_3(b)r_3(b)ru_1(c)rl_1(b)r_1(b)ru_3(b)ru_1(b) \tag{4.2}$$

と書ける。

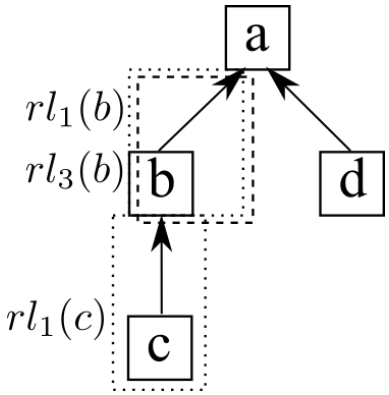

図 4.5: 二つの読み込みロック

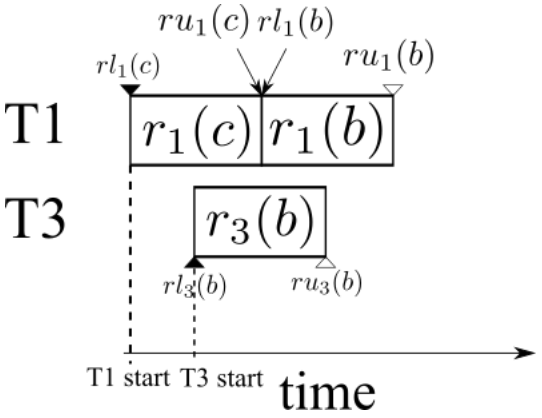

図 4.6: 二つの読み込みロックにおけるスケジュール

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*、*setTransform* の擬似コードをそれぞれアルゴリズム 4、アルゴリズム5、アルゴリズム6にて示す。アルゴリズム 4にて9〜11行目にて示されるように、木構造全体ではなく frame.rLock() で一つのフレームの読み込みロックのみ確保することにより、細粒度ロックを実装している。アルゴリズム6も同様に、一つのフレームにのみ frame.wLock() により書き込みロックを確保することにより細粒度ロックを実装している。

4.2 2PL の導入

前述した問題 2a、2b 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 (*lookupLatestTransformXact*)、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 (*setTransformsXact*) を提供して解決する。

4.2.1 データの鮮度の確保

まず、二つのフレーム間の座標変換に線形補間を行わずにフレーム間のエッジ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

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()                                  ▷ 読み込みロックを確保
10:         (trans, parent) = frame.getTransAndParent(time)
11:         frame.rUnlock()                                ▷ 読み込みロックを解放
12:         if frame == target then
13:             return source_trans
14:         end if
15:         source_trans *= trans
16:         top_parent = frame
17:         frame = parent
18:     end while
19:     frame = target
20:     target_trans = I
21:     while frame ≠ top_parent do
22:         frame.rLock()
23:         (tarns, parent) = frame.getTransAndParent(time)
24:         frame.rUnlock()
25:         if frame == source then
26:             return (target_trans)^-1
27:         end if
28:         target_trans *= trans
29:         frame = parent
30:     end while
31:     return source_trans * (target_trans)^-1
32: 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:         if  frame == target  then
13:             return common_time
14:         end if
15:     end while
16:     frame = target
17:     common_time = TIME_MAX
18:     while true do
19:         frame.rLock()
20:         (time, parent) = frame.getLatestTimeAndParent()
21:         frame.rUnLock()
22:         common_time = min(time, common_time)
23:         if parent in lct_cache then
24:             common_parent = parent
25:             break
26:         end if
27:         frame = parent
28:         if  frame == source  then
29:             return common_time
30:         end if
31:     end while
32:     for (time, parent) in lct_cache do
33:         common_time = min(common_time, time)
34:         if parent == common_parent then
35:             break
36:         end if
37:     end for
38:     return common_time
39: end function
```

Algorithm 6 細粒度ロックを実装した setTransform

```
1: procedure SETTRANSFORM(transform)
2:     frame = getFrame(transform.child_frame_id)
3:     frame.wLock()                          ▷ 書き込みロックを確保
4:     frame.insertData(transform)
5:     frame.wUnlock()                        ▷ 書き込みロックを解放
6: end procedure
```

木構造が図3.3、タイムラインが図3.4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フレームへエッジをたどりながら、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フレーム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フレームへのエッジは 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フレームへエッジをたどりながら、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フレーム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フレームへの座標変換と、木構造のルートフレームからフレーム d への座標変換を掛け合わせる。木構造のルートフレームからフレーム d への座標変換はフレーム d から木構造のルートフレームへの座標変換の逆変換から得られる。

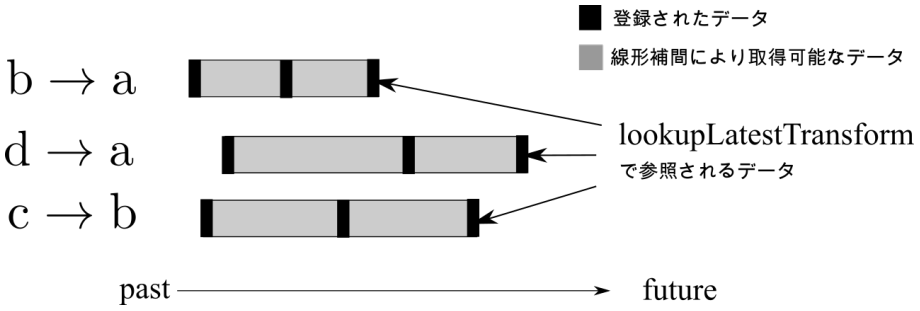

図 4.7: lookupLatestTransform で取得するデータ

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 を用いることにより、§3.6で説明した余計な負荷がかかる問題も解決できる。これは、*lookupLatestTransform* は *lookupTransform* とは異なり、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、§3.6で説明した余計な負荷がかかることはなくなるからである。

lookupLatestTransform の疑似コードをアルゴリズム7にて示す。

4.2.2 データの一貫性の確保

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

Algorithm 7 lookupLatestTransform

```
1:  function lookupLatestTransform(target, source)
2:      source_trans = I
3:      frame = source
4:      top_parent = frame
5:      while frame ≠ root do
6:          frame.rLock()
7:          (trans, parent) = frame.getLatestTransAndParent()
8:          frame.rUnlock()
9:          if frame == target then
10:             return source_trans
11:         end if
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:         if frame == source then
23:             return (target_trans)^-1
24:         end if
25:         target_trans *= trans
26:         frame = parent
27:     end while
28:     return source_trans * (target_trans)^-1
29: end function
```

うなケースでは問題となる。

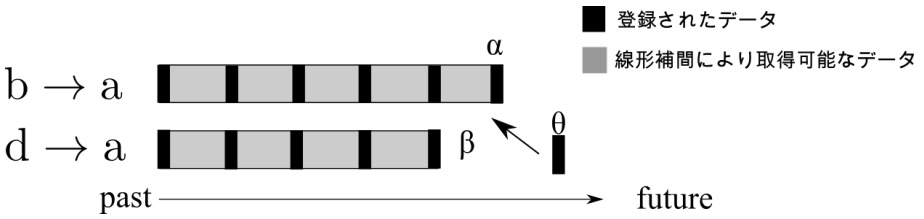

図 4.8: 同時に座標変換が登録されるケース

図4.8は木構造が図3.3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* が木構造にアクセスできることに起因する。従来の *lookupTransform* ではフレーム間のエッジ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 に木構造に登録する *setTransformsXact* を提供し、また *lookupLatestTransform* を 2PL を使うように変更した *lookupLatestTransformXact* を提供する。2PL [6] 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法である。このような性質は、並行性制御法においては Serializability [6] と呼ばれる。

2PL によって並行性制御をしたときの *setTransformsXact* と *lookupLatestTransformXact* の動作について説明する。スレッド 1 が *setTransformsXact* を用いて b→a、d→a の情報を更新し、スレッド 2 が *lookupLatestTransformXact* 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。

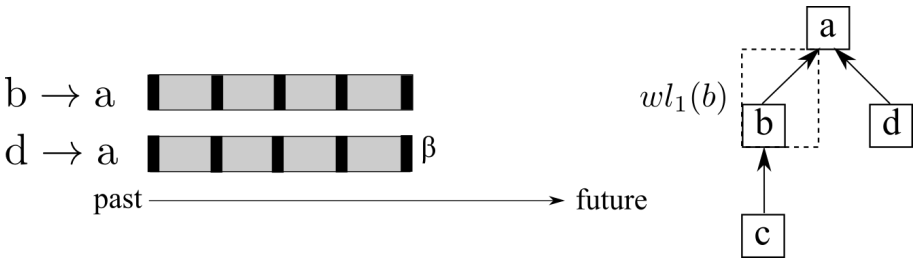

図 4.9: setTransformsXact と lookupLatestTransformXact 1

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 $wl_1(d)$

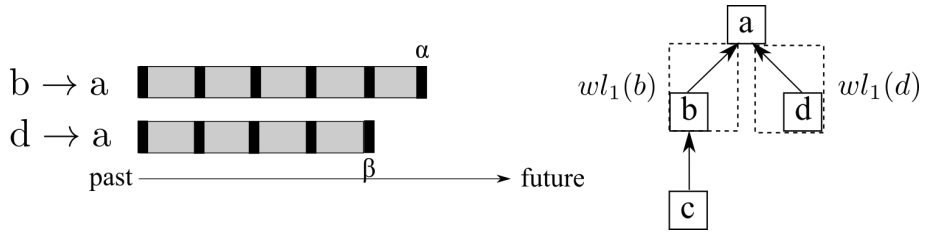

図 4.10: setTransformsXact と lookupLatestTransformXact 2

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。

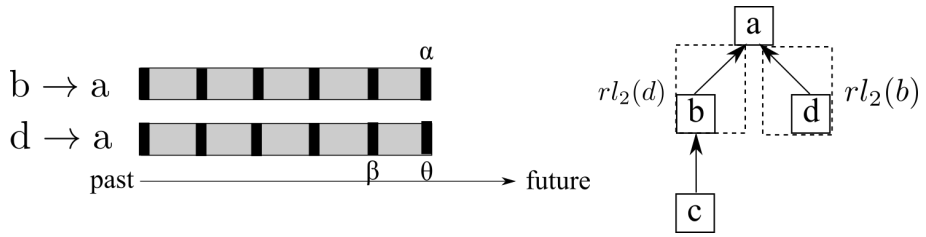

図 4.11: setTransformsXact と lookupLatestTransformXact 3

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 へのロックが解放され、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 は木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

$$wl_1(b)w_1(b)wl_1(d)w_1(d)wu_1(b)wu_1(d)rl_2(b)r_2(b)rl_2(d)r_2(d)ru_2(b)ru_2(d) \tag{4.3}$$

と表記できる。

4.2.3 Deadlock の回避

さて、2PL の導入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のような木構造において、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 *lookupLatestTransformXact* を用いて計算し、スレッド 2 が *setTransformsXact* 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め、deadlock となる。これはスレッド 2 が木構造のルートフレームへ登る方向にロックをか

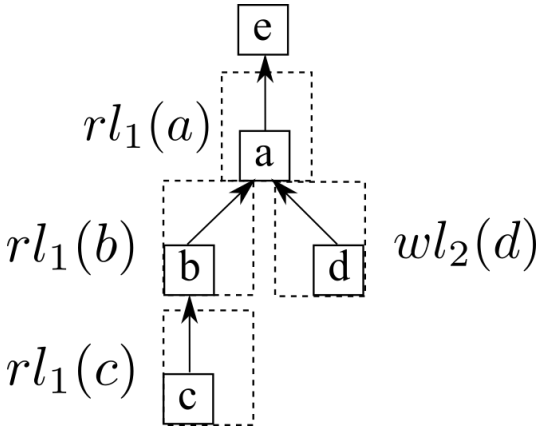

図 4.12: deadlock

けているのに対し、スレッド 1 は逆に一時的に木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 を防ぐ方法として NoWait [20] 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。

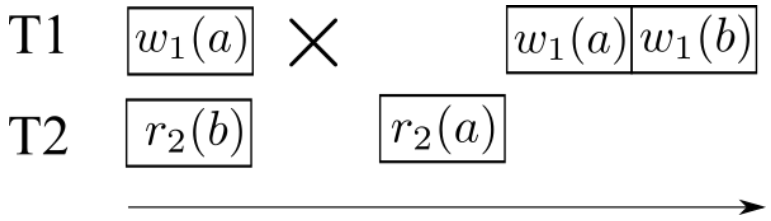

図 4.13: Dirty Read

NoWait によって処理のやり直しが発生するため、*setTransformsXact* 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 *setTransformsXact* を用いて $w_1(a)w_1(b)$、T2 が *lookupLatestTransformXact* 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。

lookupLatestTransformXact、*setTransformsXact* の擬似コードをそれぞれアルゴリズム8、9に示す。*lookupLatestTransform* とは異なり、*lookupLatestTransformXact* では読み込みロックを

取った後は8行目のように読み込みロックのリストに追加し、35行目のように座標変換を取得した後にリスト内の全てのフレームのロックを解放することにより、2PL を実装している。*setTransformsXact* も同じように書き込みロックが取れたら書き込みロックのリストに追加し、座標変換の登録ができてからリスト内の全てのフレームのロックを解放する。5行目の frame.tryWLock() はもし他のスレッドが読み込みロックも書き込みロックも確保していない場合には書き込みロックをし、true を返す。もし他のスレッドがロックを確保していた場合には false を返し、書き込みロックの確保失敗を表す。書き込みロックの確保が失敗失敗したらリスト内の全てのフレームのロックを解放し、1ms 待機してから処理をやり直す。

Algorithm 8 lookupLatestTransformXact

```
1: function lookupLatestTransformXact(target, source)
2:     rlock_list = [ ]
3:     source_trans = I
4:     frame = source
5:     top_parent = frame
6:     while frame ≠ root do
7:         frame.rLock()
8:         rlock_list.push_back(frame)                    ▷ 読み込みロックのリストに追加
9:         (trans, parent) = frame.getLatestTransAndParent()
10:        if frame == target then
11:            for f in rlock_list do                     ▷ リスト内の全ての要素のロックを解放
12:                f.rUnlock()
13:            end for
14:            return source_trans
15:        end if
16:        source_trans *= trans
17:        top_parent = frame
18:        frame = parent
19:    end while
20:    frame = target
21:    target_trans = I
22:    while frame ≠ top_parent do
23:        frame.rLock()
24:        rlock_list.push_back(frame)                    ▷ 読み込みロックのリストに追加
25:        (tarns, parent) = frame.getTransAndParent(time)
26:        if frame == source then
27:            for f in rlock_list do                     ▷ リスト内の全ての要素のロックを解放
28:                f.rUnlock()
29:            end for
30:            return (target_trans)^-1
31:        end if
32:        target_trans *= trans
33:        frame = parent
34:    end while
35:    for f in rlock_list do                             ▷ リスト内の全ての要素のロックを解放
36:        f.rUnlock()
37:    end for
38:    return source_trans * (target_trans)^-1
39: end function
```

Algorithm 9 setTransformsXact

```
1:  procedure setTransformsXact(transforms)
2:      wlock_list = [ ]
3:      for trans in transforms do
4:          frame = getFrame(trans.child_frame_id)
5:          lock_success = frame.tryWLock()                    ▷ 書き込みロックの確保を試みる
6:          if lock_success then
7:              wlock_list.push_back(frame)
8:          else
9:              for f in wlock_list do                         ▷ リスト内の全ての要素のロックを解放
10:                 f.wUnlock()
11:             end for
12:             sleep 1ms
13:             goto 2                                         ▷ 処理をやり直す
14:         end if
15:     end for
16:     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
17:         frame = getFrame(trans.child_frame_id)
18:         frame.insertData(trans)
19:     end for
20:     for f in wlock_list do                                 ▷ リスト内の全ての要素のロックを解放
21:         f.wUnlock()
22:     end for
23: end procedure
```

第5章　評価

TF ライブラリに細粒度を実装した *lookupTransform*・*setTransform* インターフェイス、及び最新の複数のデータを atomic に取得・更新できる *lookupLatestTransformXact*・*setTransformsXact* インターフェイスを、一回のインターフェイスの呼び出しを 1 タスクとし、以下の指標で評価した。

1. スループット: 一秒間に何回タスク (*lookupTransform* 及び *setTransforms*) ができたか
2. レイテンシ: タスクの応答時間
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 (*lookupLatestTransformXact* のみ): *lookupLatestTransformXact* 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 (*setTransformsXact* のみ): *setTransformsXact* において、NoWait によってタスクをやり直した回数の比率。(やり直した回数 / *setTransformsXact* 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 *lookupLatestTransformXact*・*setTransformsXact* を latest と呼称する。

5.1　実装

§1.2で述べたように、TF ライブラリは ROS 上で動作する。ROS は Ubuntu 上で動作し、Ubuntu のバージョンごとに別のディストリビューションが公開されている。これに合わせ、TF ライブラリも ROS のディストリビューション毎に提供されている。TF ライブラリの実装は Github のリポジトリ [21] で公開され、ブランチ毎に各 ROS のディストリビューション向けの実装がされている。この対応関係を表5.1に示す。

Ubuntu のバージョン	対応する ROS のディストリビューション	対応する geometry2 のブランチ名
20.04(Focal)	ROS Noetic Ninjemys	noetic-devel
18.04(Bionic)	ROS Melodic Morenia	melodic-devel
14.04(Trusty)	ROS Indigo Igloo	indigo-devel

表 5.1: Ubuntu のバージョンと対応するブランチ名

このリポジトリのデフォルトブランチは melodic-devel であるが、TF の木構造の並行性制御アルゴリズムはどのブランチでも変わらない。また、ROS2 向けの TF ライブラリの実装は [22] で公開されているが、こちらも木構造の並行性制御アルゴリズムは ROS 向けのものと変わらない。

このため、本研究では Ubuntu18.04 を搭載したマシンに ROS Melodic Morenia をインストールし、TF ライブラリの Github のリポジトリ [21] の melodic-devel の実装を変更して実験を行った。C++言語で実装をし、1236 行分の変更を行った。この実装は [23] で公開されている。

5.2 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た。

5.3 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3.10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* 呼び出しで読みこむフレームの数。joint 個のフレームのうち一様分布を元にランダム (つまり、Skew=0) 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一様分布を元にランダムに 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。
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。

ここで、*setTransform* では write_len 個のデータの書き込みにおいて、write_len 回 *setTransform* を呼び出すので write_len タスク実行できたとみなすが、*setTransformsXact* では write_len 個のデータの書き込みでは、一回の呼び出しで write_len 個のデータの書き込みができるので、1 タスク実行できたとみなすことに注意する。つまり、*setTransformsXact* の方がスループット・レイテンシの評価において不利になる。各インターフェイスの呼び出しに対応するタスク数を表5.2に示す。

インターフェイス	一度の呼び出しでアクセスするフレーム数	タスク数の数え方
old の lookupTransform	read_len	read_len 個のフレームを読むのが 1 タスク
old の setTransform	1	1 個のフレームに書き込むのが 1 タスク
snapshot の lookupTransform	read_len	read_len 個のフレームを読むのが 1 タスク
snapshot の setTransform	1	1 個のフレームに書き込むのが 1 タスク
latest の lookupLatestTransformXact	read_len	read_len 個のフレームを読むのが 1 タスク
latest の setTransformsXact	write_len	write_len 個のフレームに書き込むの が 1 タスク

表 5.2: 各インターフェイスとタスクの数え方

各実験はそれぞれ YCSB-A/B/C [19] ワークロードについて行った。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。また、上述のように Skew は 0 に設定して実験を行った。

5.4 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。old と比べ、論理コア数である 224 倍以上の性能差が出たのは、次に説明するように TF の木構造の C++の実装において各フレームを管理する方法を既存手法から変更したからだと考えられる。

各フレームは C++の実装において TimeCache クラスで表現され、TF の木構造中のフレーム群は std::vector<std::shared_ptr<TimeCache> >で管理される。std::shared_ptr 型は自身を参照しているスコープの数をカウンタで管理するため、複数スレッドから std::shared_ptr 型のデータにアクセスする際にこのカウンタへの読み込み・書き込みが複数スレッドから行われる。これによりキャッシュミス率が増加し性能劣化につながる。これを避けるため、提案手法の実装ではフレーム群は std::vector<TimeCache*>で管理される。このような実装の違いが論理コア数である 224 倍以上の性能差につながったと考えられる。

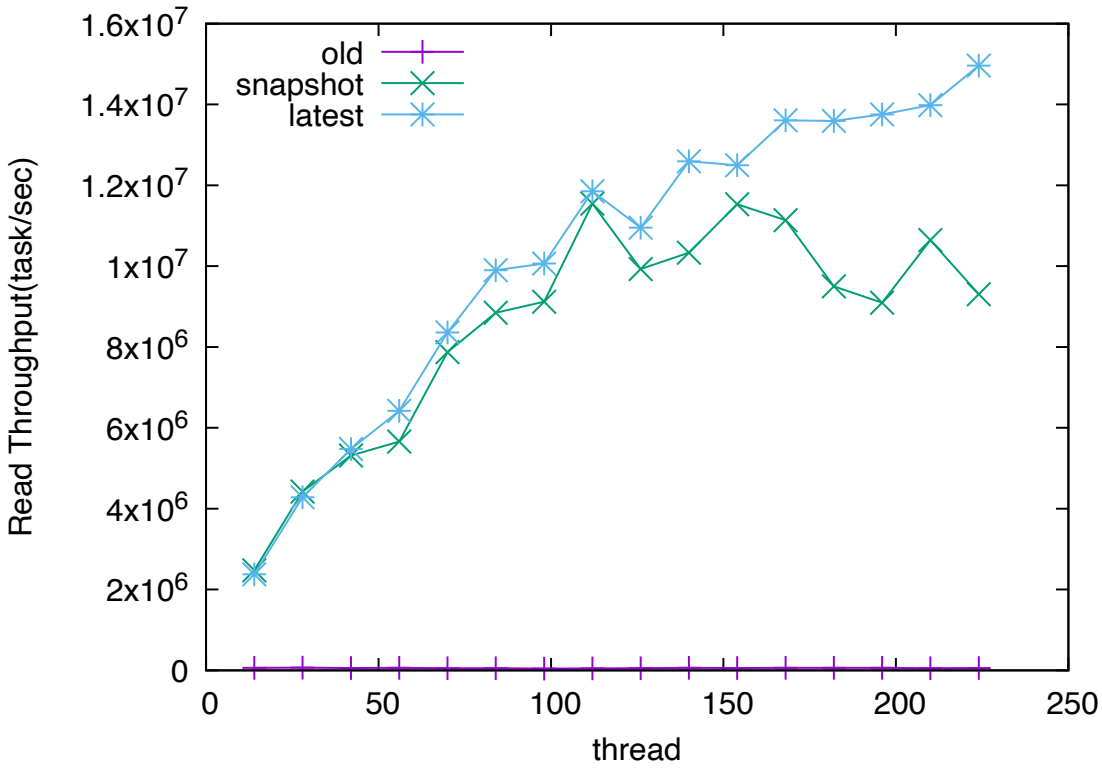

図 5.1: YCSB-C におけるスレッド数と読み込みスループットの関係

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、§3.4で説明したように木構造を 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 率については記述しない。

5.5 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

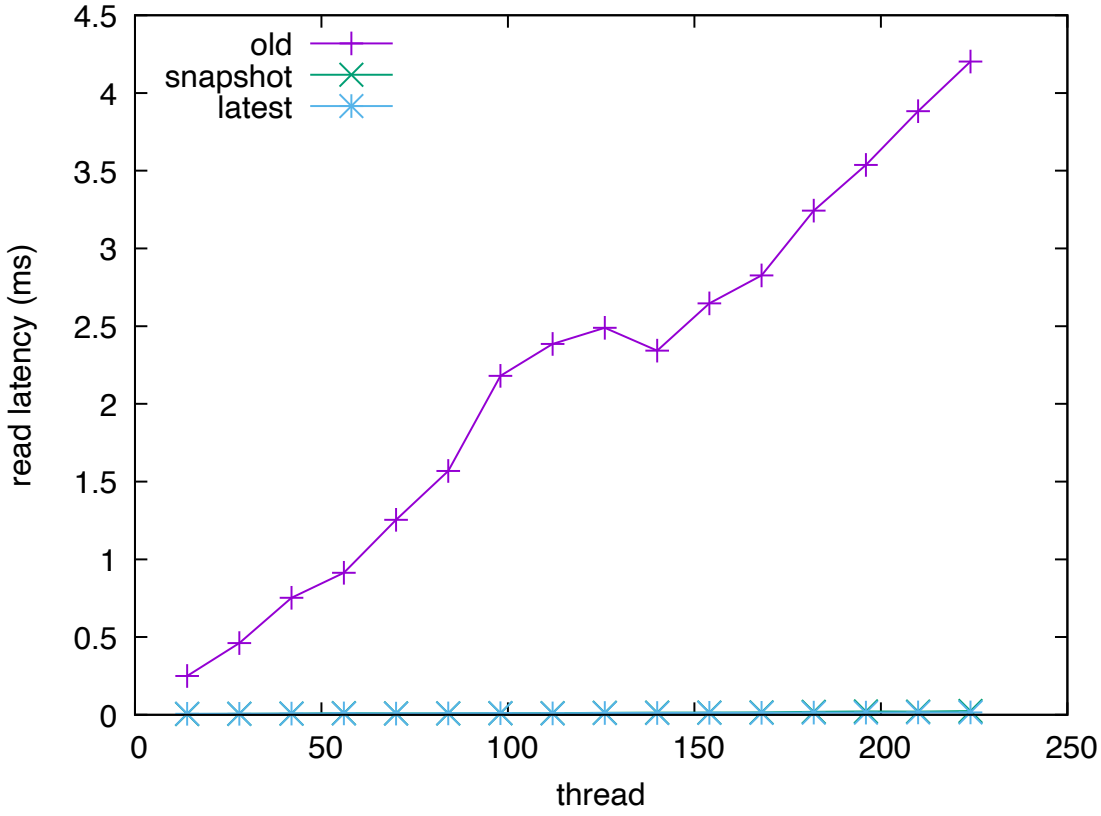

図 5.2: YCSB-C におけるスレッド数とレイテンシの関係

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る。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。

§3.5にて説明したように、各フレームに登録された座標変換データは 10 秒間保存される。これは C++の std::deque という両端キューで実装されており、snapshot の *lookupTransform* では二回目の各フレームへのアクセスにおいて、この両端キュー内の要素を二部探索することにより、該当する時刻の座標変換データを取得する [24]。このキュー内の要素へのアクセスも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。latest の *lookupLatestTransformXact* では両端キューの先頭にのみアクセスするので、キュー内の要素へのアクセスは発生しない。

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増加によるものだと考えられる。

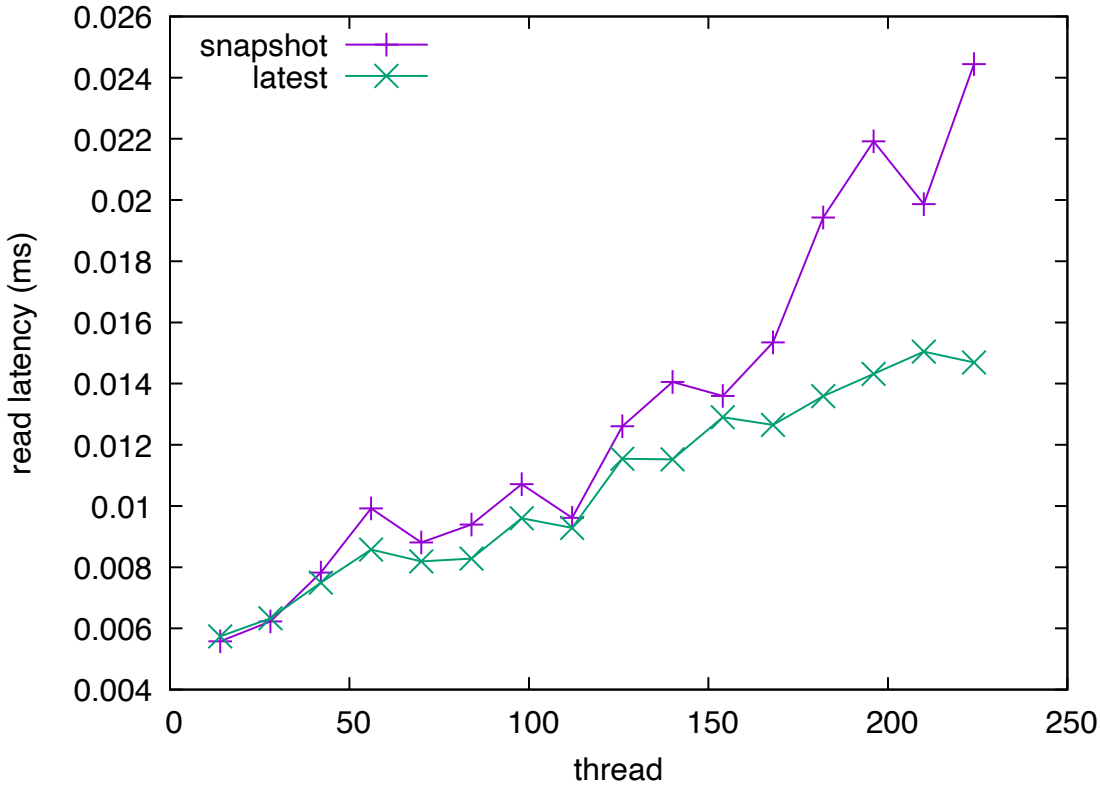

図 5.3: YCSB-C におけるスレッド数とレイテンシの関係 snapshot と latest のみ

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。

5.6 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§5.5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 の *setTransformsXact* 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、snapshot の *setTransform* では一つの要素の書き込み及び書き込みロックを行えばよく、さらに§5.5で説明したようなキャッシュミス率の変化も発生して

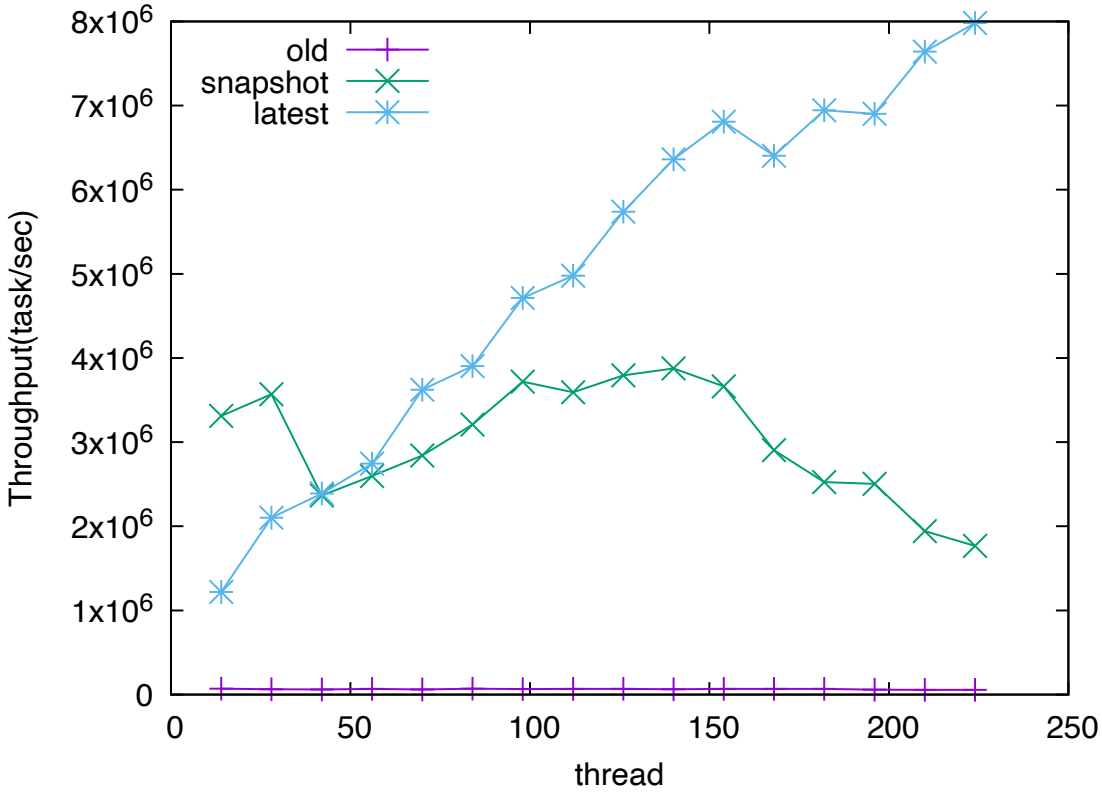

図 5.4: YCSB-A におけるスレッド数とスループットの関係

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。

5.7 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。

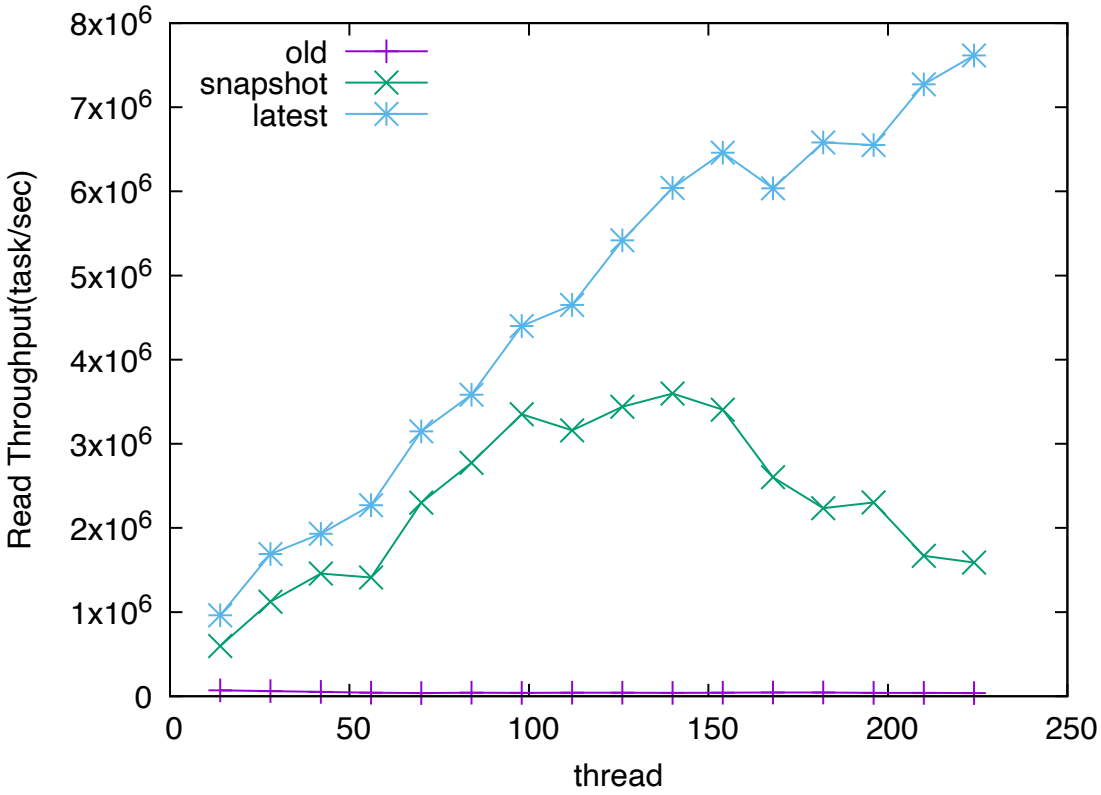

図 5.5: YCSB-A におけるスレッド数と読み込みスループットの関係

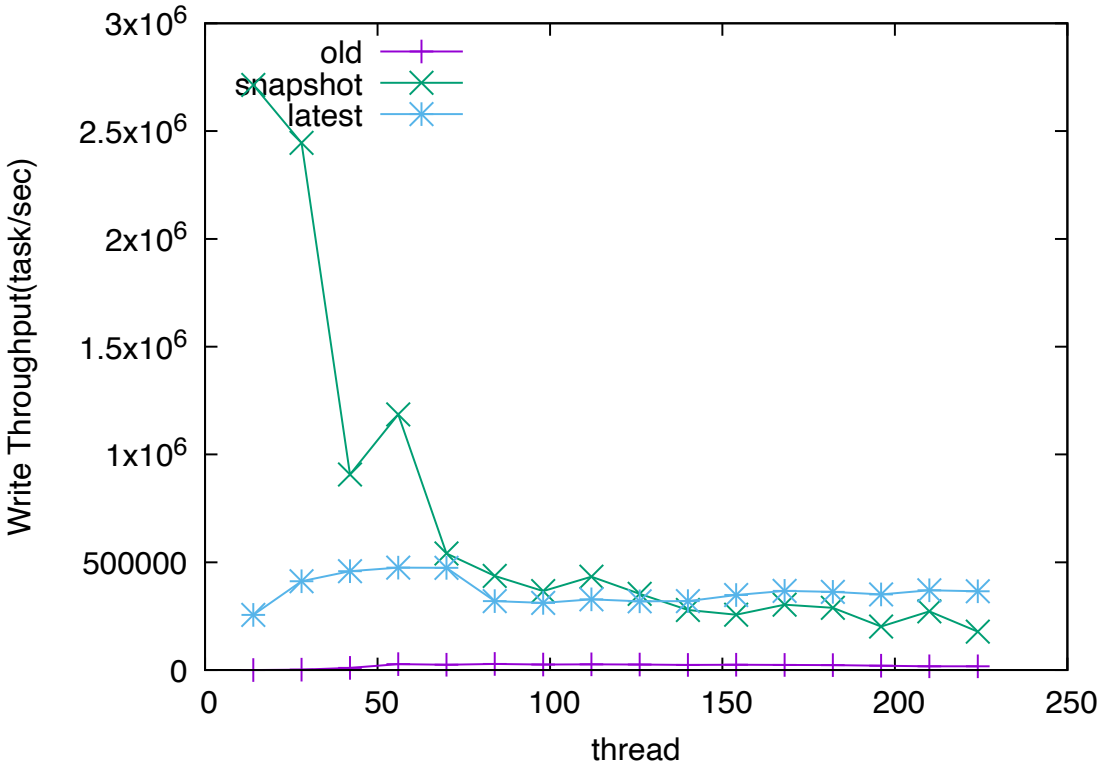

図 5.6: YCSB-A におけるスレッド数と書き込みスループットの関係

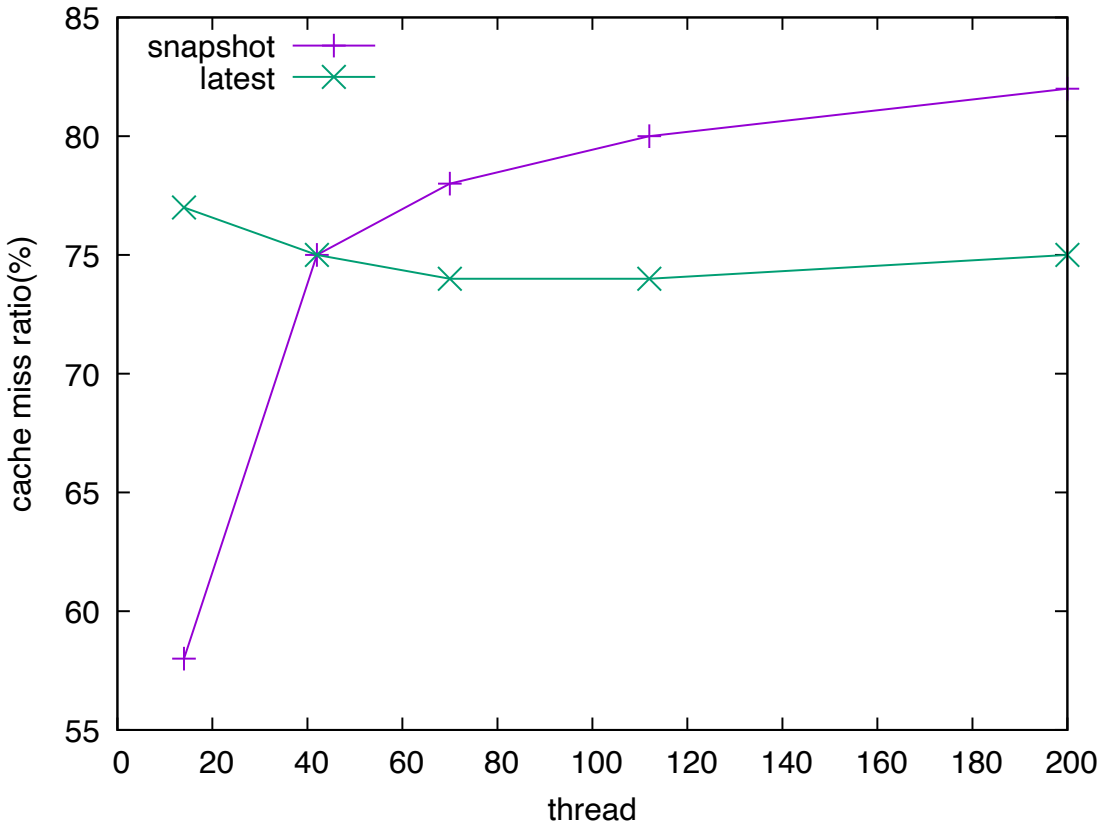

図 5.7: YCSB-A におけるスレッド数とキャッシュミス率の関係

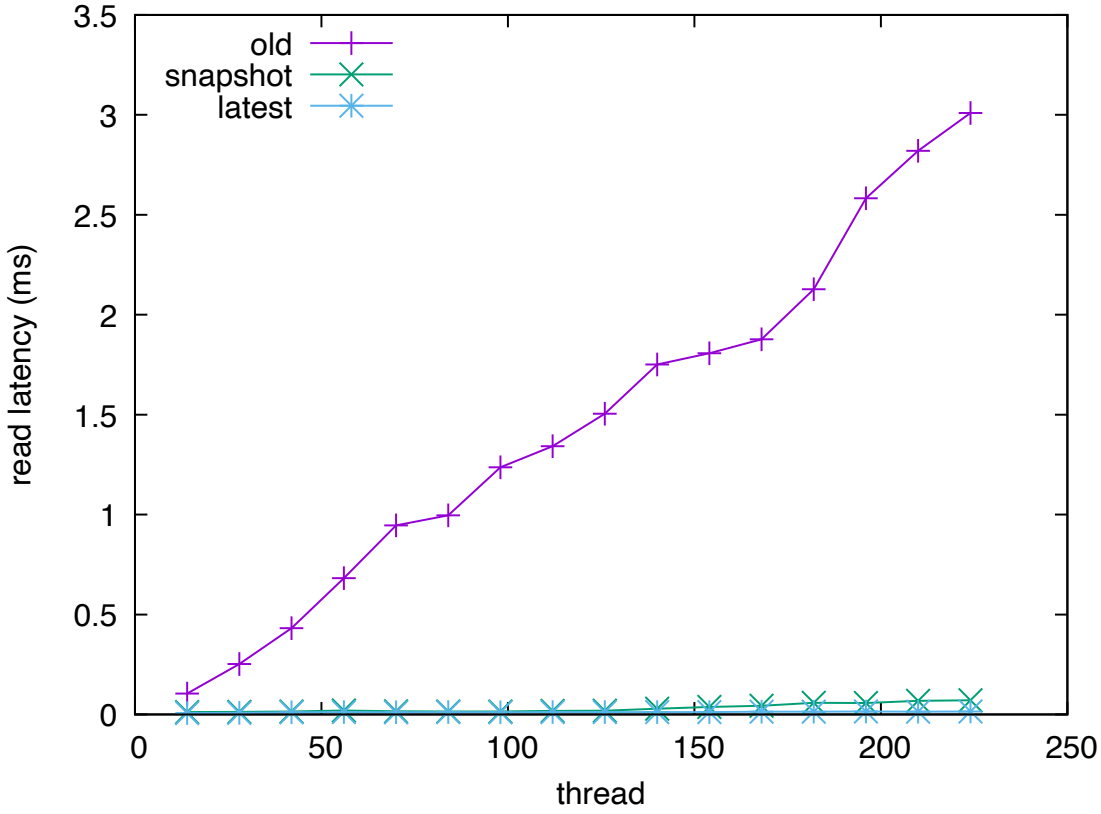

図 5.8: YCSB-A におけるスレッド数と読み込みレイテンシの関係

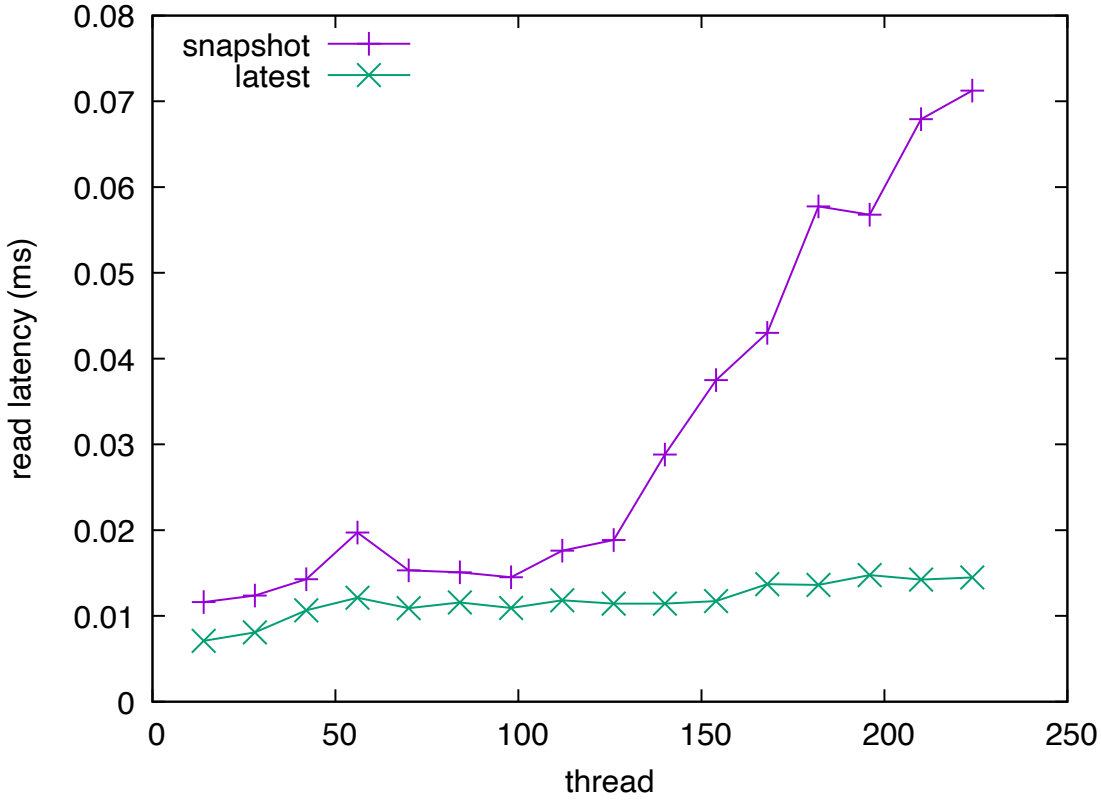

図 5.9: YCSB-A におけるスレッド数と読み込みレイテンシの関係 snapshot と latest のみ

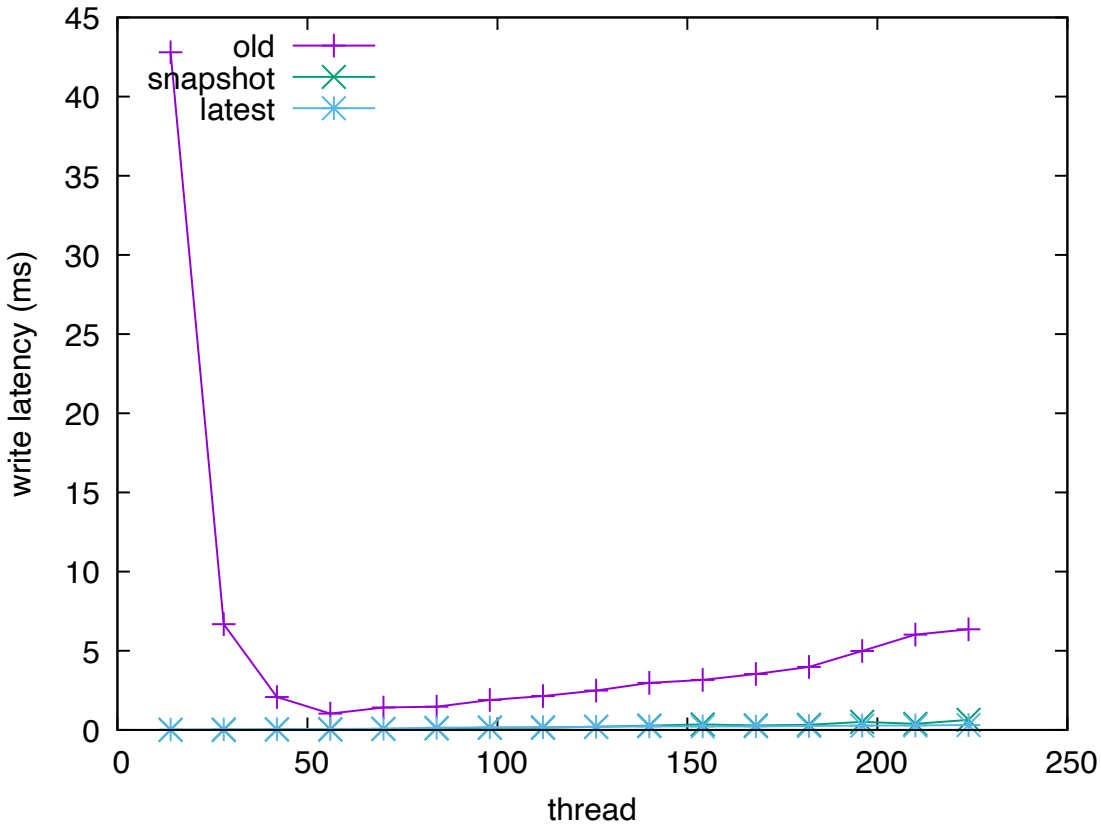

図 5.10: YCSB-A におけるスレッド数と書き込みレイテンシの関係

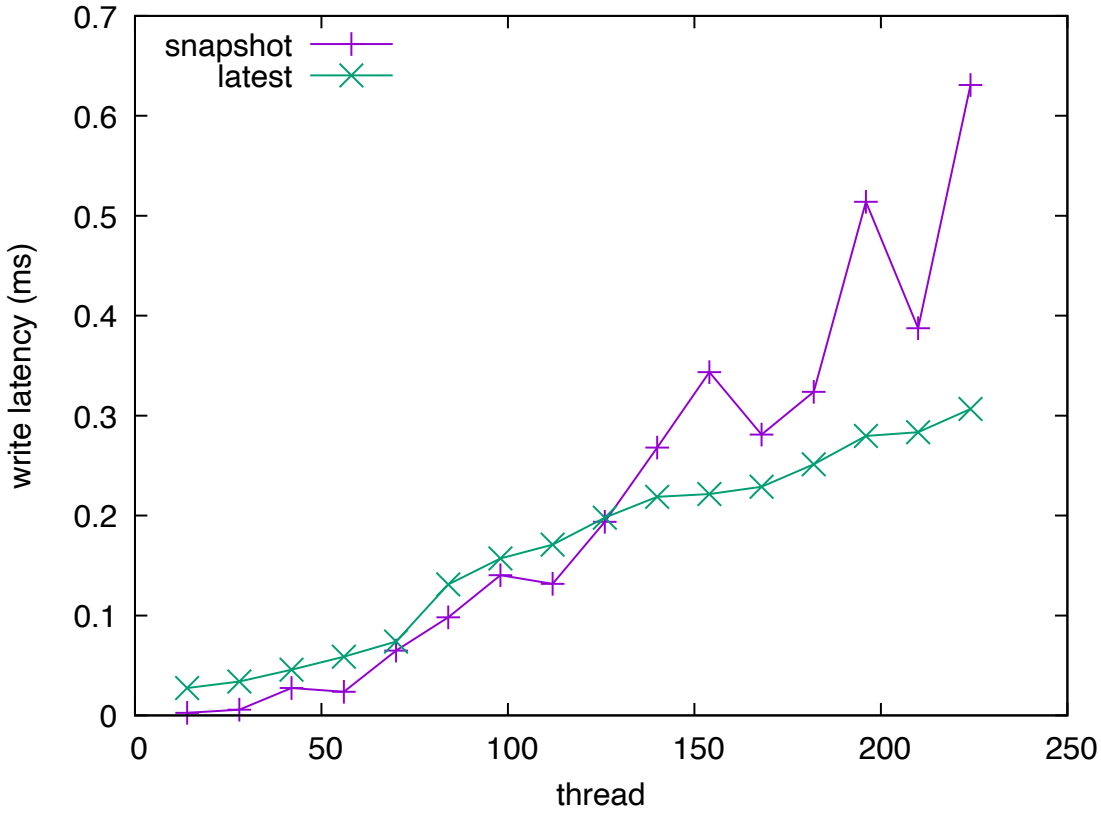

図 5.11: YCSB-A におけるスレッド数と書き込みレイテンシの関係 snapshot と latest のみ

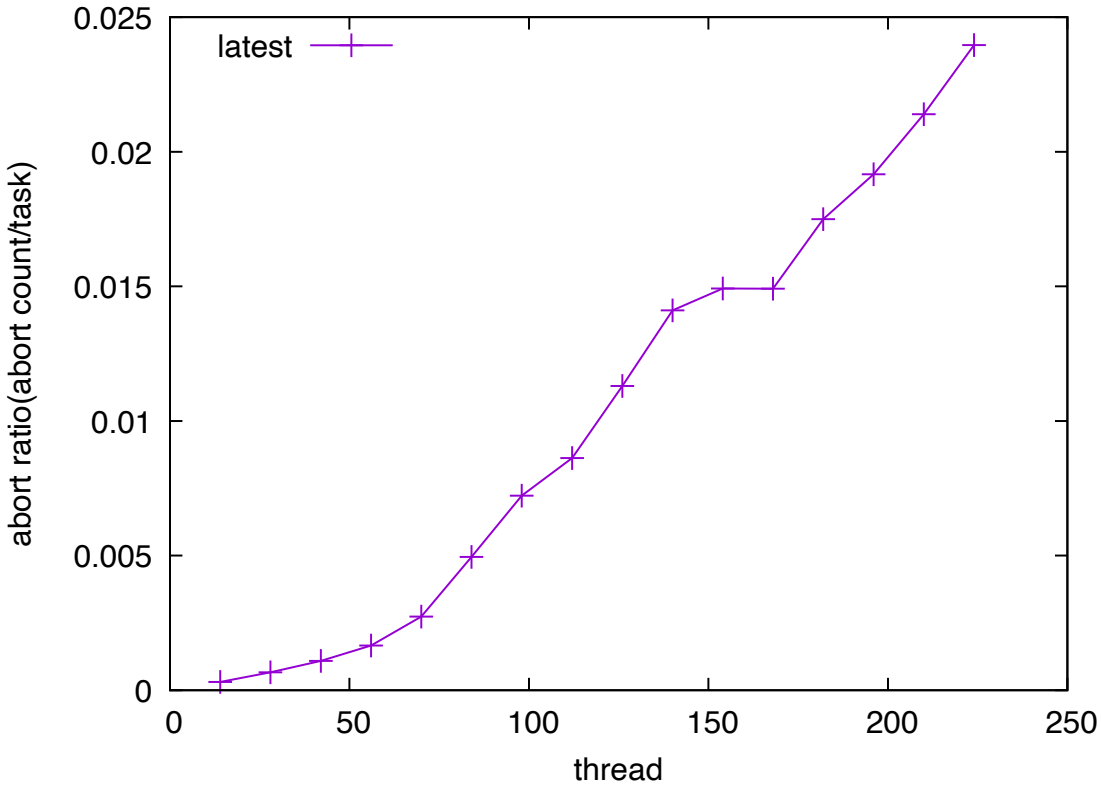

図 5.12: YCSB-A におけるスレッド数と abort 率の関係

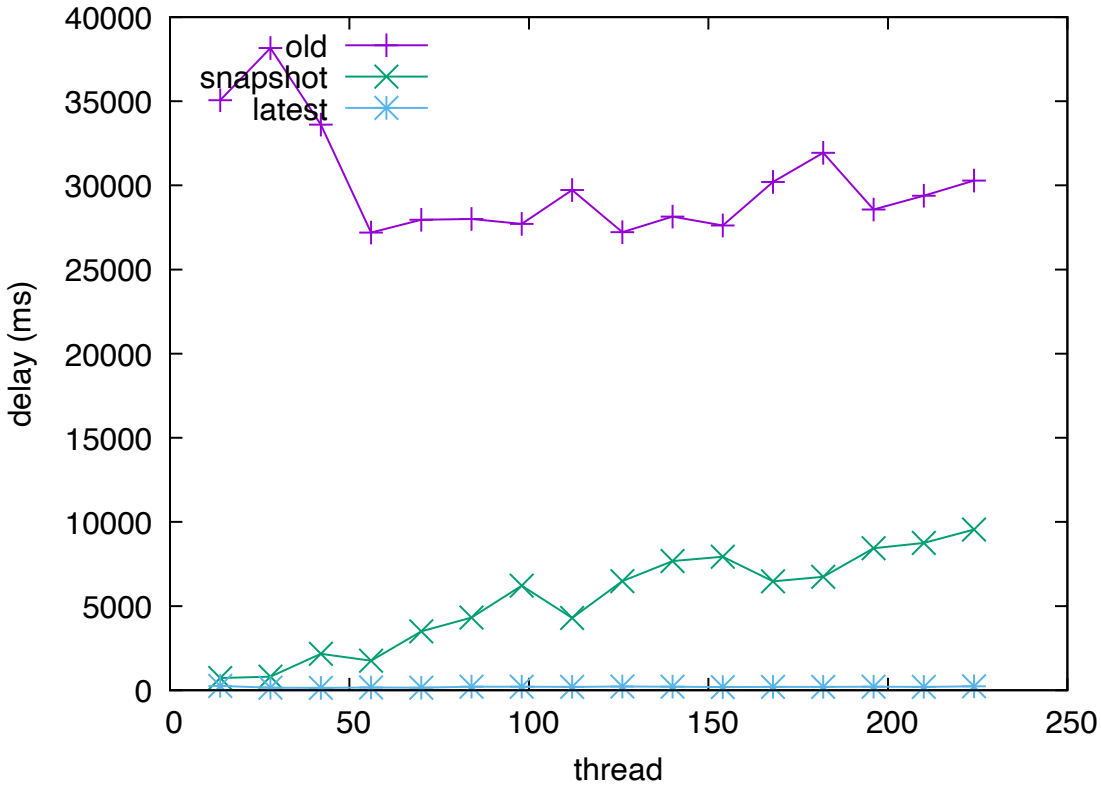

図 5.13: YCSB-A におけるスレッド数とデータの鮮度の関係

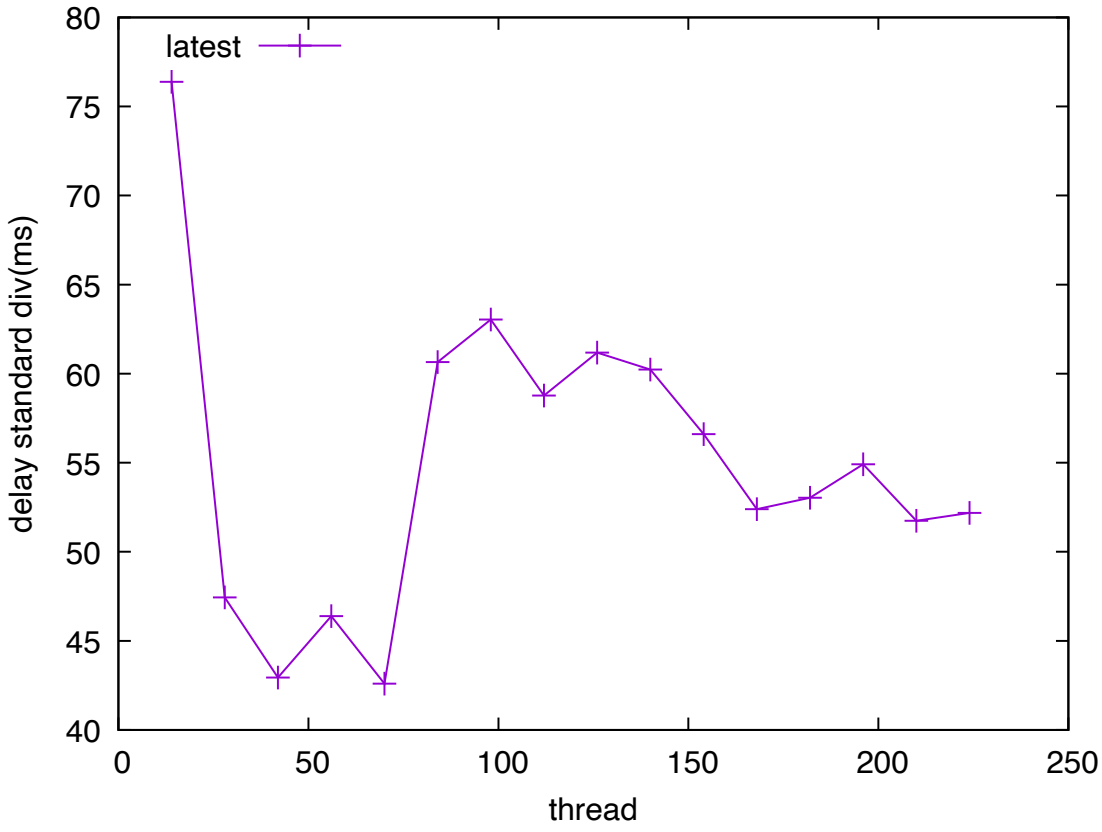

図 5.14: YCSB-A におけるスレッド数とデータの同期性の関係

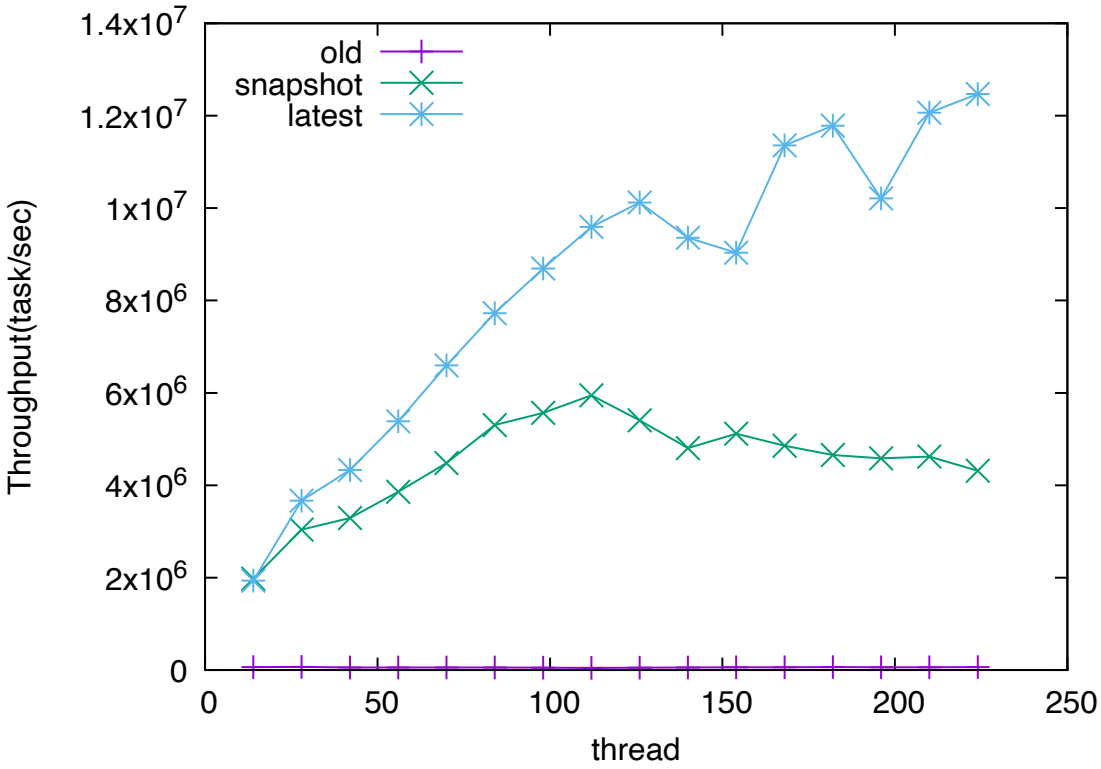

図 5.15: YCSB-B におけるスレッド数とスループットの関係

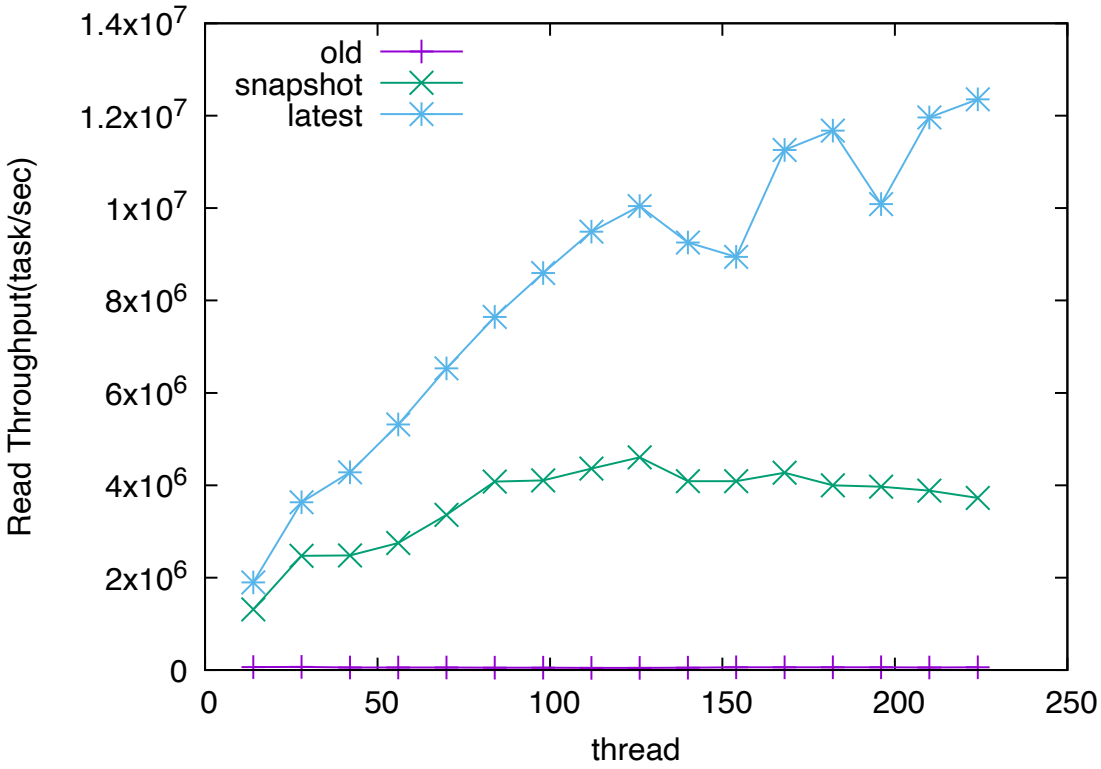

図 5.16: YCSB-B におけるスレッド数と読み込みスループットの関係

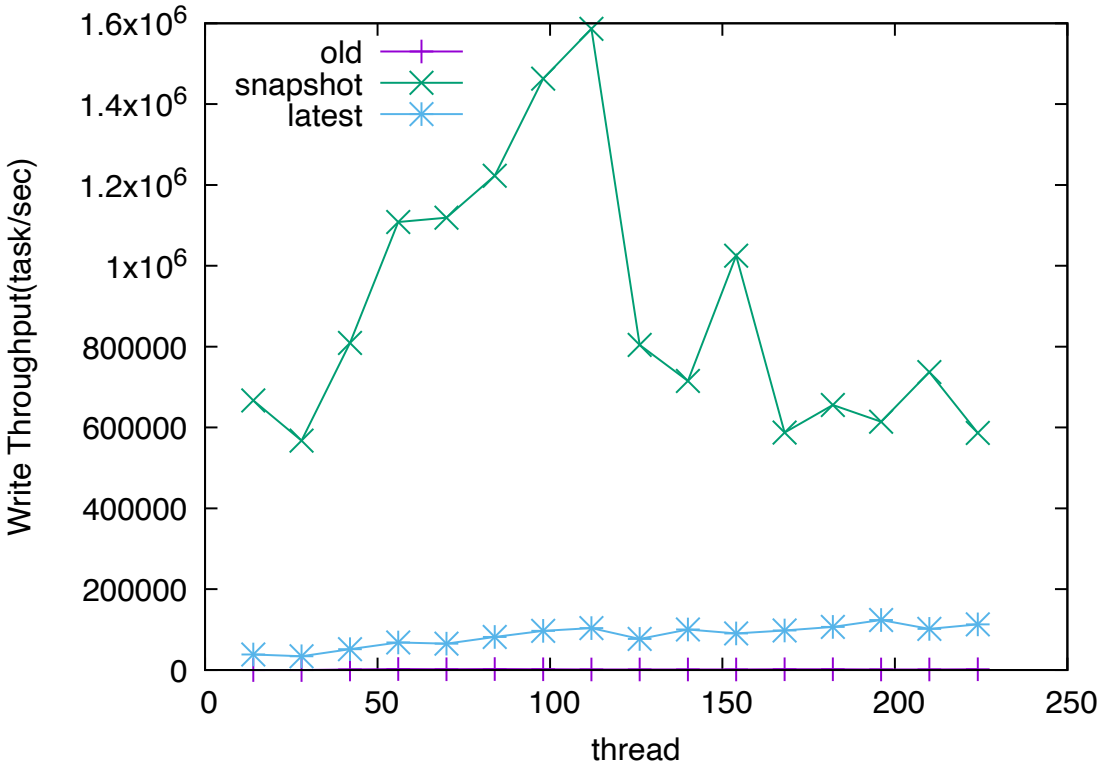

図 5.17: YCSB-B におけるスレッド数と書き込みスループットの関係

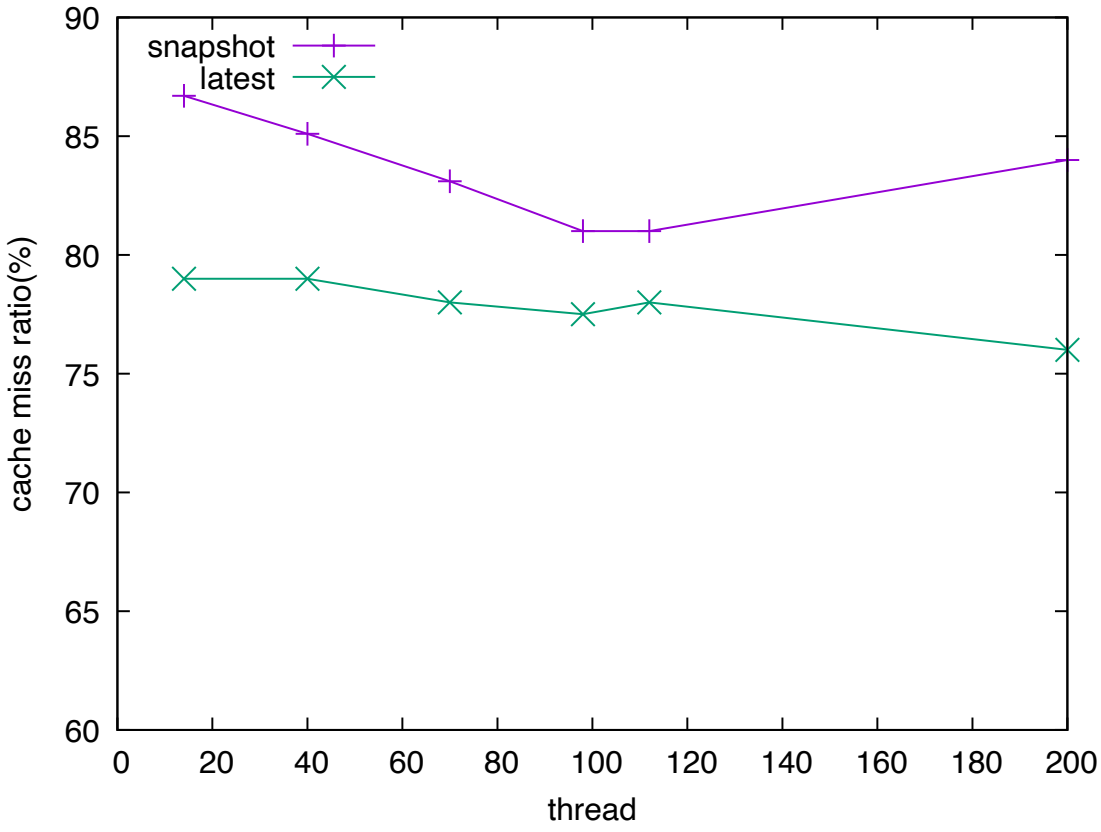

図 5.18: YCSB-B におけるスレッド数とキャッシュミス率の関係

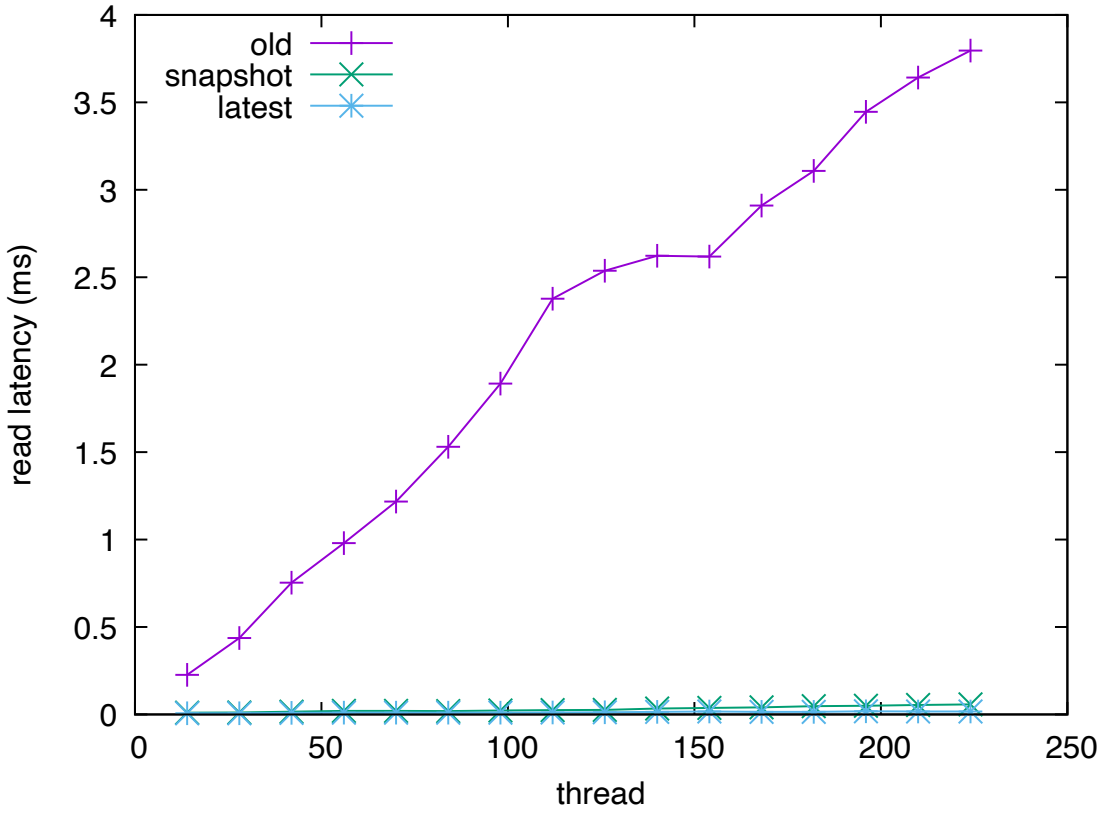

図 5.19: YCSB-B におけるスレッド数と読み込みレイテンシの関係

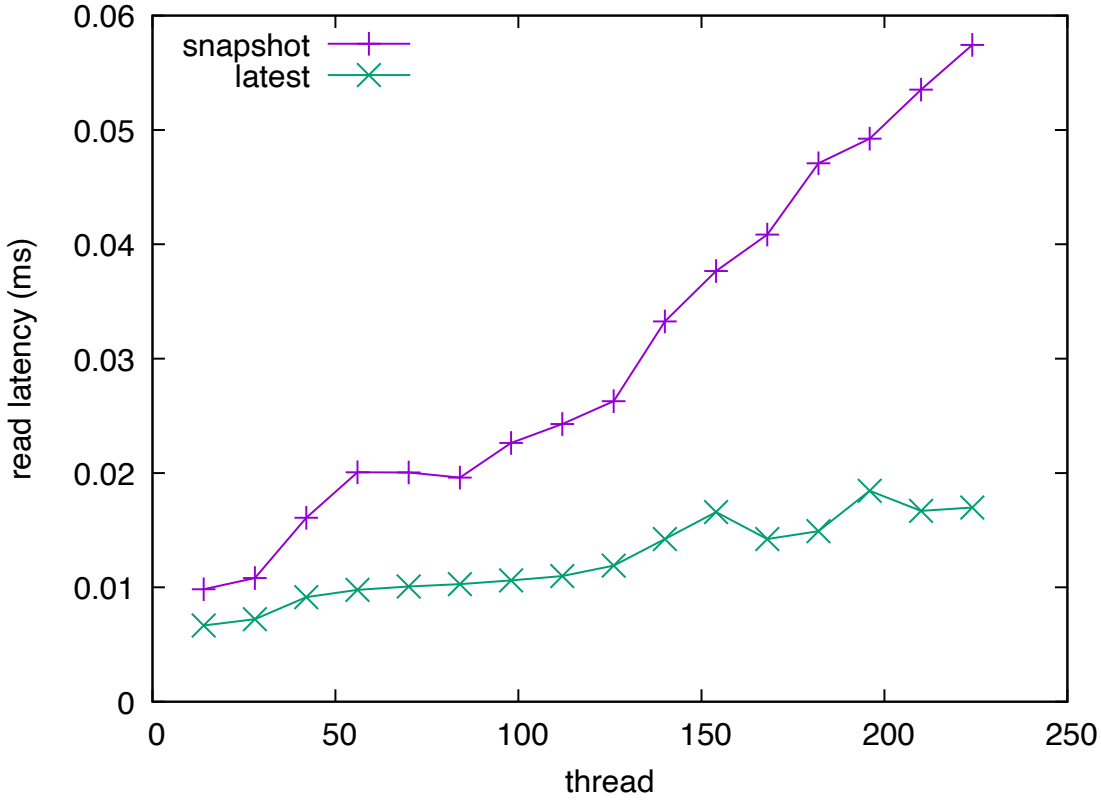

図 5.20: YCSB-B におけるスレッド数と読み込みレイテンシの関係 snapshot と latest のみ

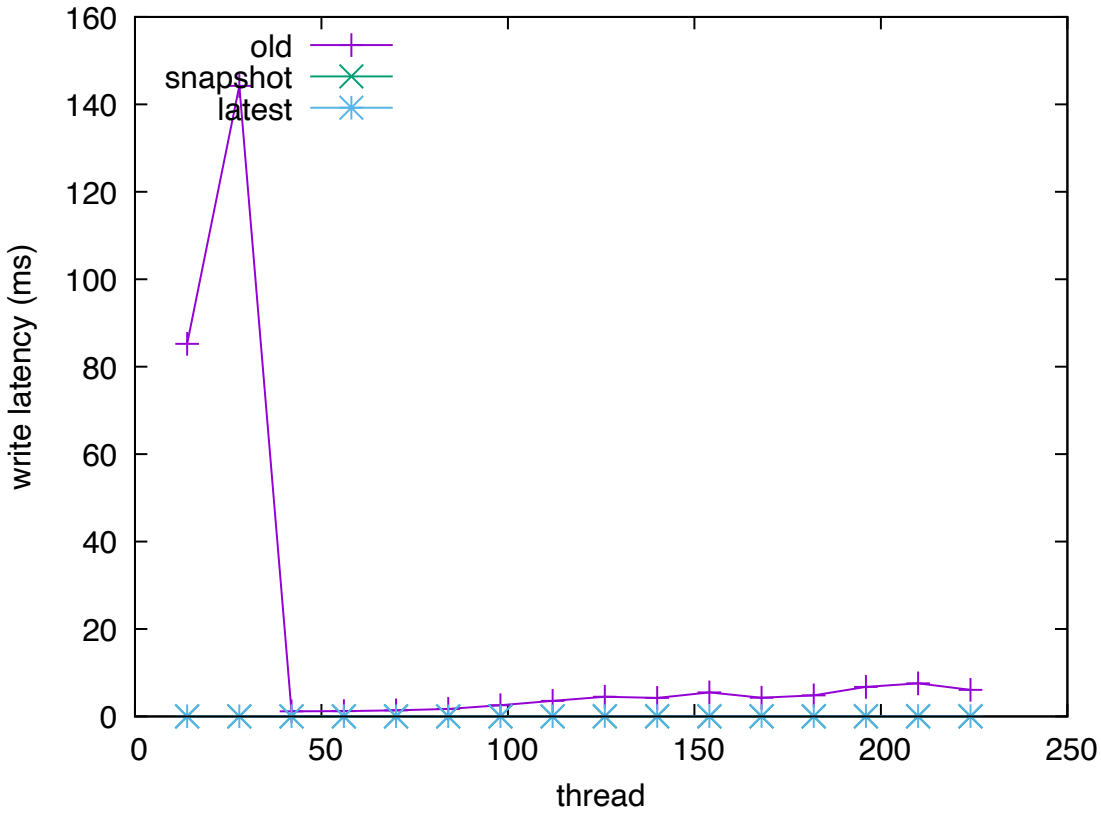

図 5.21: YCSB-B におけるスレッド数と書き込みレイテンシの関係

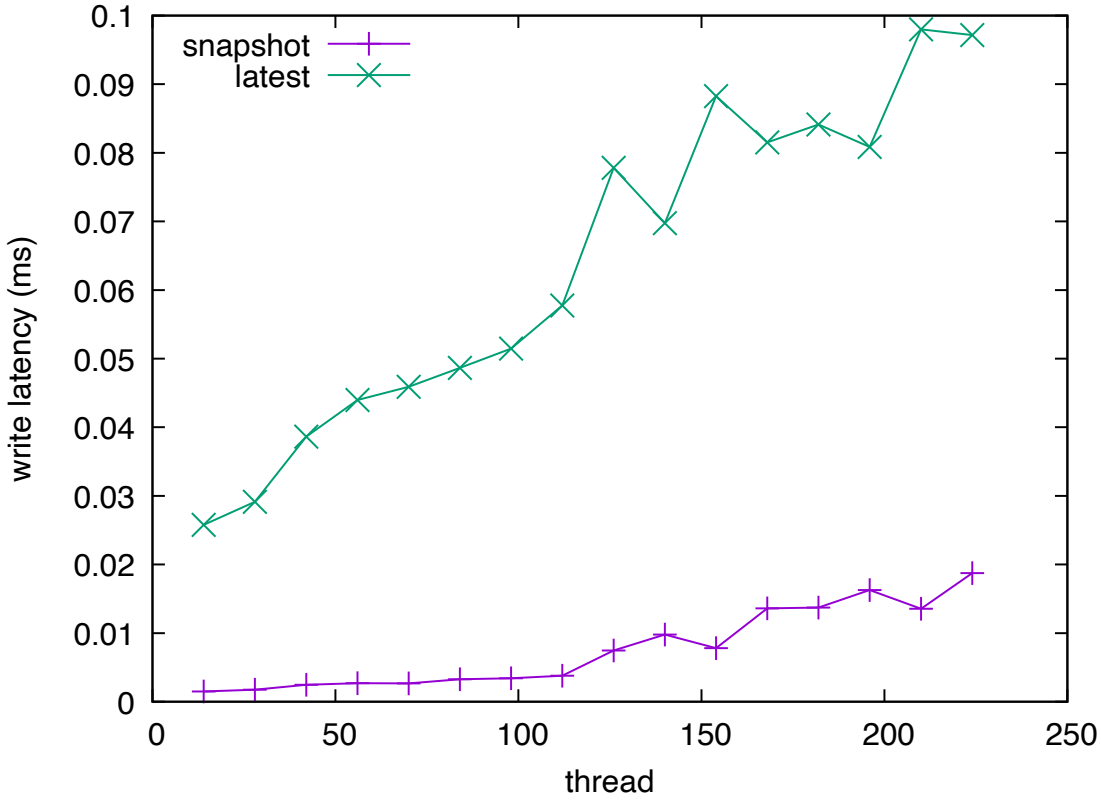

図 5.22: YCSB-B におけるスレッド数と書き込みレイテンシの関係 snapshot と latest のみ

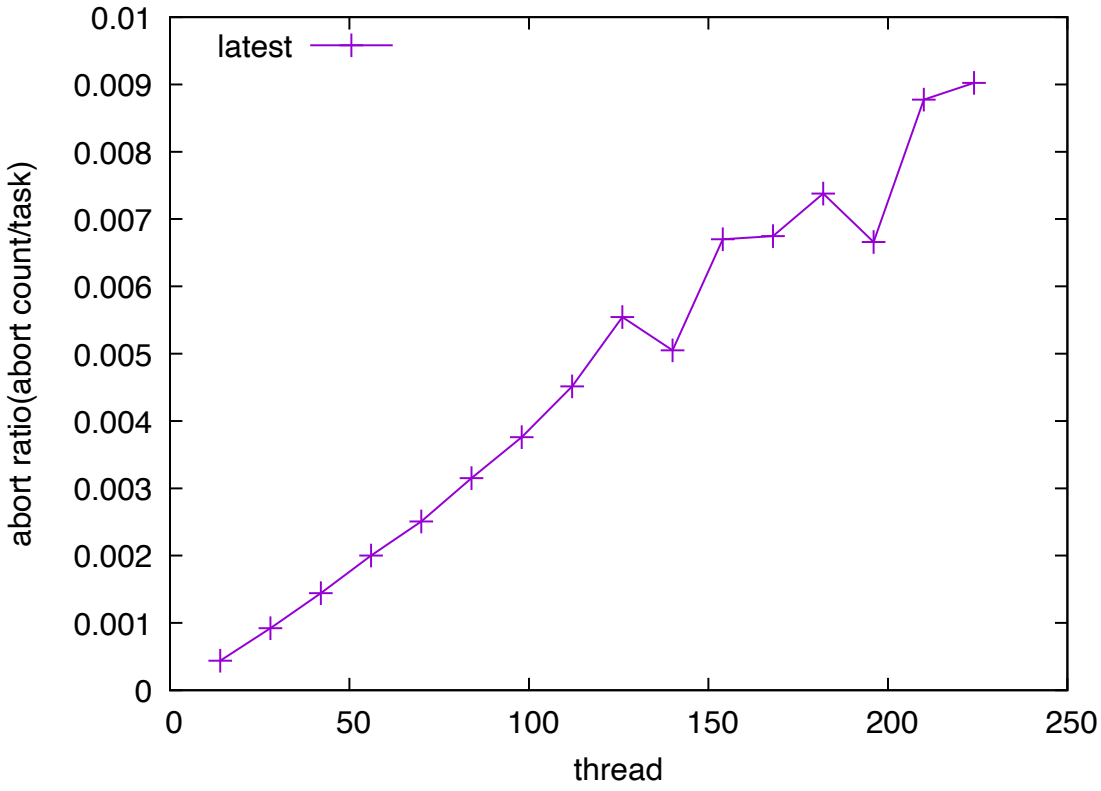

図 5.23: YCSB-B におけるスレッド数と abort 率の関係

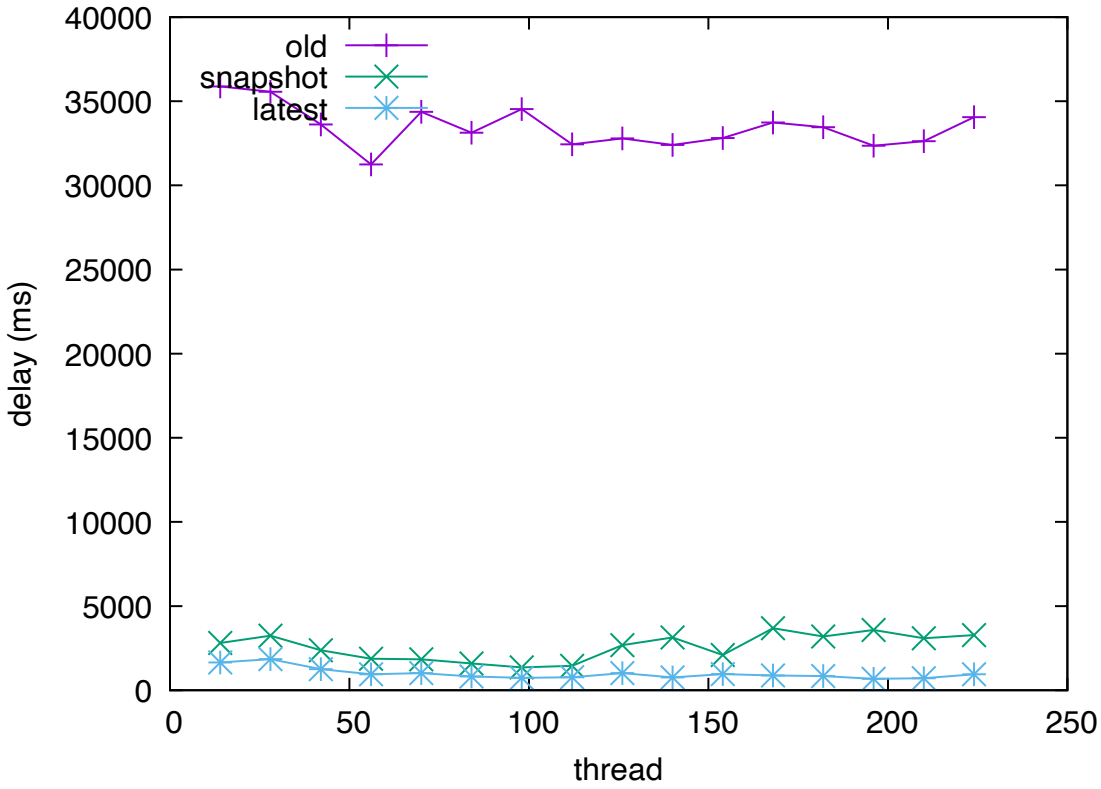

図 5.24: YCSB-B におけるスレッド数とデータの鮮度の関係

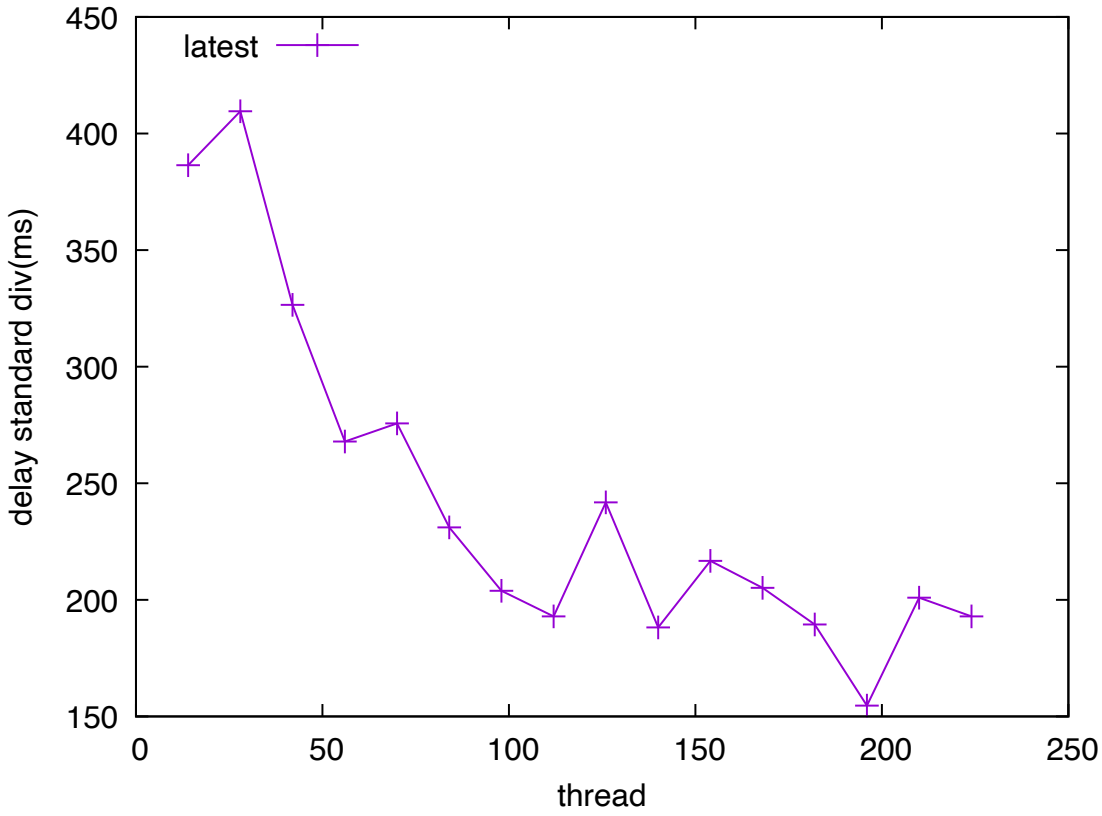

図 5.25: YCSB-B におけるスレッド数とデータの同期性の関係

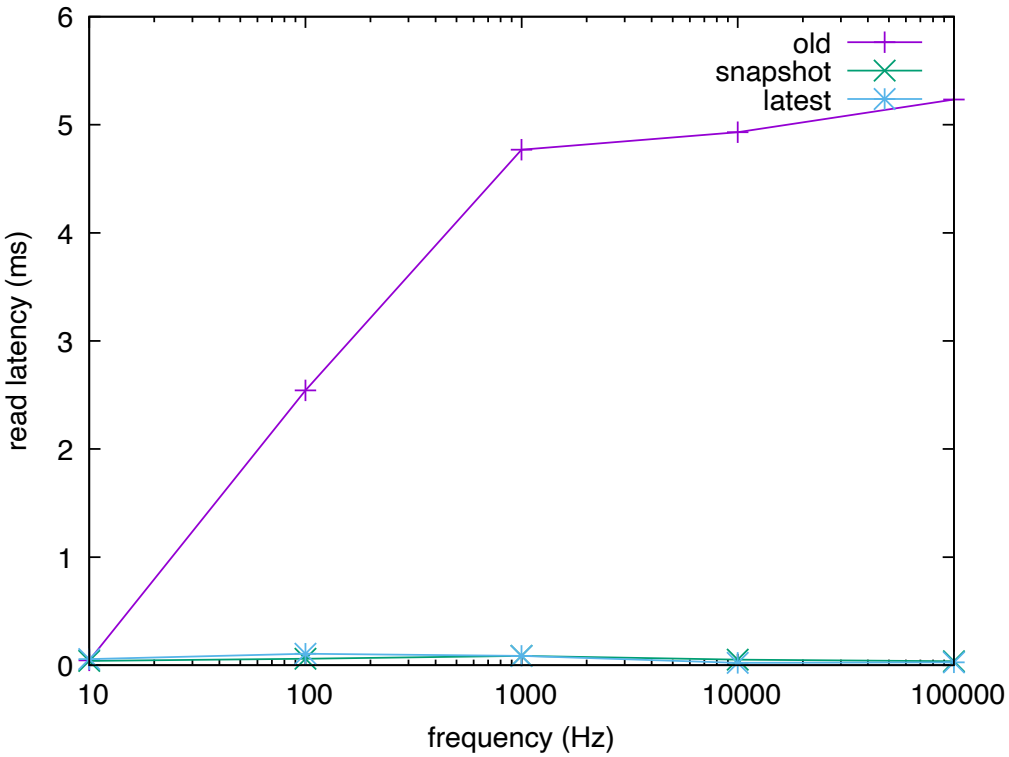

図 5.26: 制御周期とレイテンシ

第6章　議論

本研究のアプリケーションとして次の三つが挙げられる。

- 関節数が多く、高精度な制御が求められるヘビ型ロボット
- 自動運転におけるエッジクラウド
- 大量のドローンの群の制御

自動運転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。

[25] には各都道府県における各自動車道 (高速自動車国道、一般国道の自動車専用道路、一般国道など) の 12 時間あたりの交通量が記載されており、どの道路でも 12 時間あたりの交通量は高々80,000 である。[26] には全国の高速道路の一日の交通量が記載されており、東北自動車道が 277,584 で最大である。これらのデータを元に、高速道路の入り口 (IC) から出口 (別の IC) までのローカルな区域の情報を一つのエッジサーバーが管理すると想定すると、次のようなワークロードが想定できる。

- エッジクラウドにて管理する自動車の台数は高々1000 台。
- 各自動車は、自分の周辺の自動車や障害物、標識など、高々20 の情報を書き込む。
- 各自動車は、渋滞回避や障害物などの回避のために高々100 の周辺の物体の情報を読み込む。

本研究の評価において行った実験と同じように、joint=1000,、read_ratio=0.5、read_len=100、write_len=20、frequency=120 に設定してこのワークロードを再現すると、既存手法、提案手法の読み込みレイテンシは図6.1、6.2のようになった。スレッド数が 224 の状況において既存手法では読み込みレイテンシは 16ms 程度となった。frequency=120 で設定してあるので読み込みレイテンシは約 8.3ms 以内にする必要があるが、既存手法ではこのデッドラインに間に合っていないことがわかる。これに対し、提案手法の読み込みレイテンシは高々0.24ms 程度となり、デッドラインに間に合っていることがわかる。

自動運転におけるエッジクラウドにて TF ライブラリを使用する際には、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・削除による大量のフレームの追加・削除
- 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

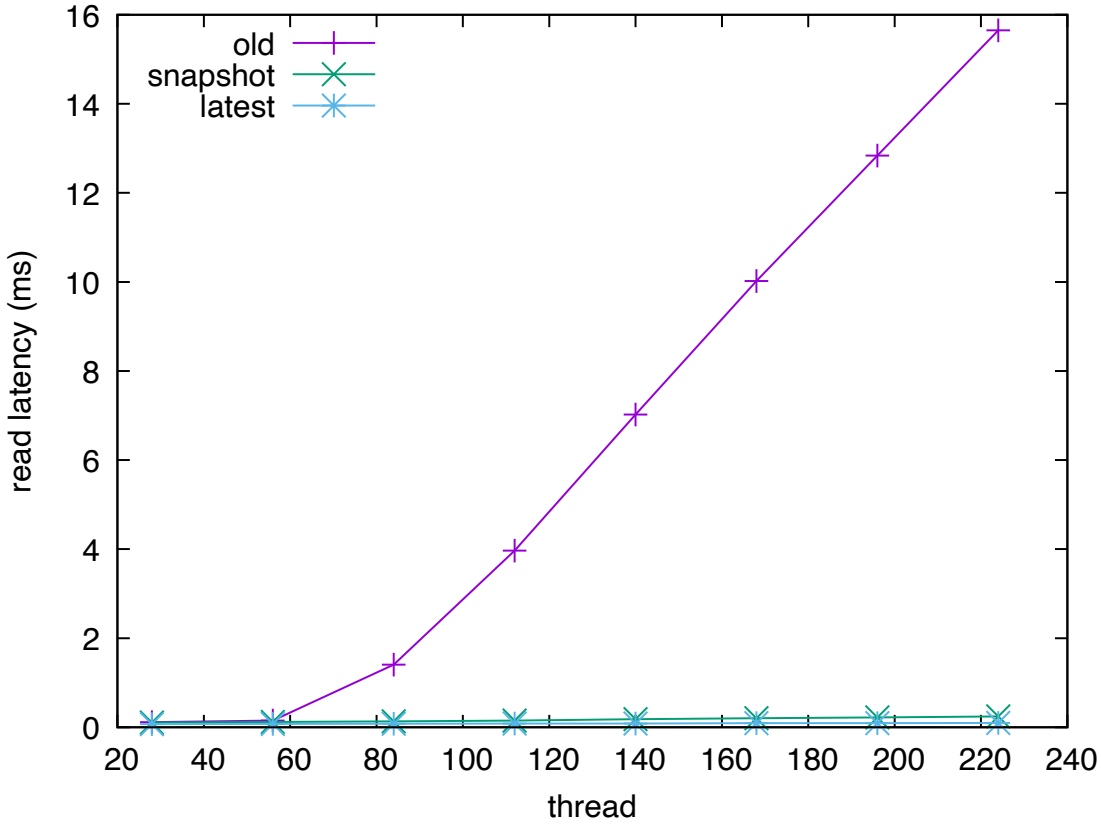

図 6.1: 自動運転におけるエッジクラウドのワークロードの再現

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。

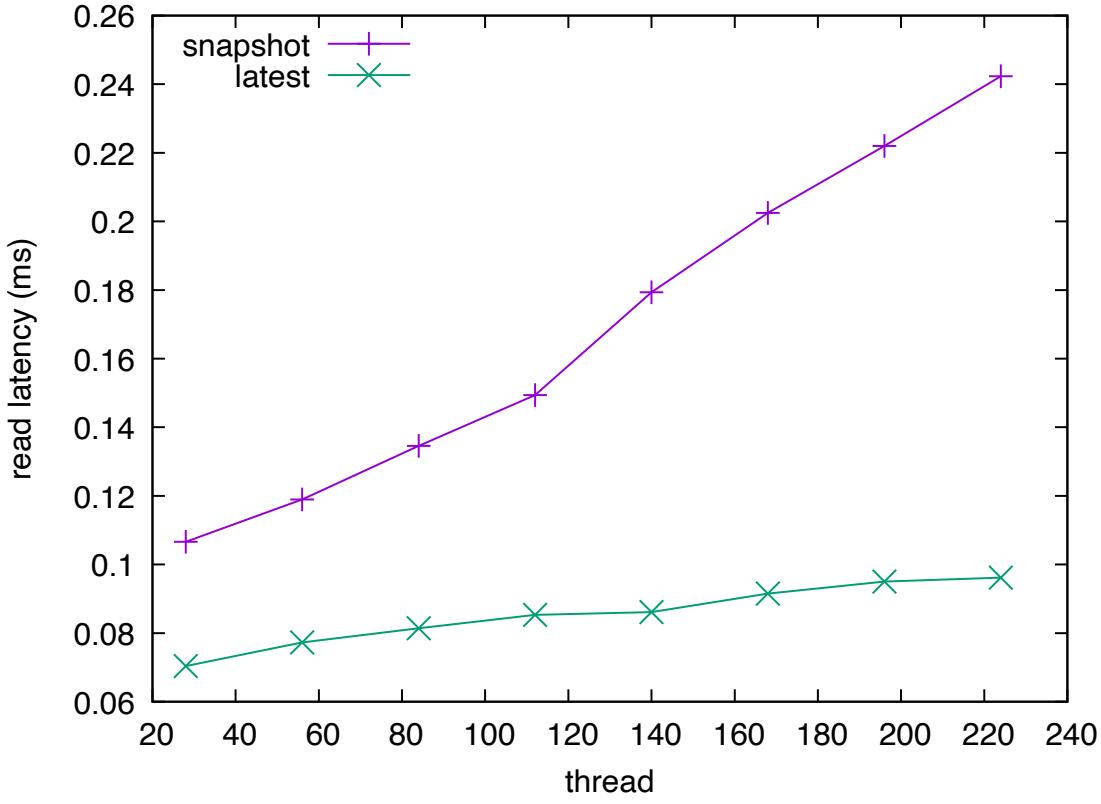

図 6.2: 自動運転におけるエッジクラウドのワークロードの再現 snapshot と latest のみ

第7章　結論

7.1　結論

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法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法における 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 (*lookupLatestTransformXact*)、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 (*setTransformsXact*) 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事を示した。

7.2　今後の課題

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法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法の導入が必要である事を示した。

本研究では TF ライブラリの木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法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法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法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] "自律移動型ロボットの世界市場は 2027 年まで年平均成長率 19.6 %で成長すると予想される", https://www.jiji.com/jc/article?k=000004775.000067400&g=prt

[2] "IDC Japan ／自律移動型ロボットが 2023 年に 561 億円の市場規模に", https://www.lnews.jp/2019/05/l0514309.html

[3] 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.

[4] 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.

[5] 竹内栄二朗 (2008). 移動ロボットの基本機能のモジュール化と環境地図生成に関する研究, 筑波大学博士 (工学) 学位論文.

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

[7] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for high- ly-contended dynamic workloads on a thousand cores. Proceedings of the VLDB En- dowment, Vol. 10, No. 2, p. p. 49âĂŞ60, 2016.

[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] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, Osamu Tatebe: An Analysis of Concurrency Control Protocols for In-Memory Databases with CCBench. Proc. VLDB Endowment 13(13): 3531-3544 (2020).

[11] 南祐衣, 鈴木汰一, 山倉拓海, 構勇海, 潘鴻遠, 及川裕介 (筑波大), 荻原湧志, 川島英之 (慶應大), 伊達央, 萬礼応 (筑波大): つくばチャレンジ 2021 における 筑波大学知能ロボット研究室チーム Aqua の取り組み. 第 22 回　計測自動制御学会.

[12] 伊達央, 阿部太郎: ヘビ型ロボットのオドメトリと遠隔操作支援. 第 8 回横幹連合コンファレンス.

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

[14] "BufferCore.cpp lookupTransform", https://github.com/ros/geometry2/blob/melodic-devel/tf2/include/tf2/buffer_core.h

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

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

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

[18] "ndt_matching", https://github.com/Autoware-AI/core_perception/tree/master/lidar_localizer/nodes/ndt_matching

[19] 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.

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

[21] "ros/geometry github repository", https://github.com/ros/geometry2

[22] "ros2/geometry2 github repository", https://github.com/ros2/geometry2

[23] "Ogiwara-CostlierRain464/geometry2 github repository", https://github.com/Ogiwara-CostlierRain464/geometry2

[24] "cache.cpp", https://github.com/Ogiwara-CostlierRain464/geometry2/blob/68651682b124cb9d3403cf729e8f9e0c1c1319f5/tf2/src/cache.cpp#L166

[25] "NEXCO 東日本 高速道路の通行台数と料金収入（令和2年度", https://www.e-nexco.co.jp/activity/word_data/data/r02.html

[26] "平成 27 年度　全国道路・街路交通情勢調査 一般交通量調査　集計表 交通量整理表（都道府県別道路種別別", https://www.mlit.go.jp/road/census/h27/data/pdf/syuukei04.pdf

[27] Dean De Leo and Peter Boncz. Teseo and the Analysis of Structural Dynamic Graphs. Proc. VLDB Endowment 13(13): 3531-3544 (2020).