修士論文の和文要旨

大学院情報システム学研究科　博士前期課程		情報システム基盤学　専　攻
氏　名	村井 栄王	学籍番号　1253019
論文題目	ロックとSTMを共存させた排他制御機構の提案	

要　旨

排他制御はマルチスレッドシステムの開発効率上，性能上のボトルネックとなっている．

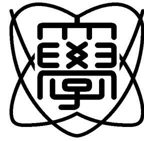
従来はロックを使用して排他制御を行うのが主流である．この手法は書き込みが多い状況では比較的高速な排他制御を行うことができるが，読み込みが多い状況では比較的遅い．また，ロックの粒度が細かい場合はデッドロックの危険性があり，プログラマビリティが低い．

そのプログラマビリティの問題を解決するために，Touitouらが提案したのがソフトウェアトランザクショナルメモリ（Software Transactional Memory: STM）である．これはデータベースのトランザクションに真似て排他制御を行うものである．STMでは，トランザクションは投機的に共有メモリへのアクセスを行い，データの矛盾が発生したことに気づくと排他制御のはじめに戻る（ロールバック）．STMの実装によってはロックを使用しない．また，ロックベースSTMの実装ではデッドロックを回避する機能が備わっているため，プログラマはデッドロックを意識することなくアプリケーションを開発することができ，プログラマビリティが高い．また，STMではリードロックを獲得する必要がなく，コア数だけ共有メモリの読み込みを行うことができるので，読み込みが多いなどのロールバックが起きにくい状況ではロックより高速に動作することができる．しかし，書き込みが多い状況だと逆にロックよりも性能が劣る．

本研究では，マルチスレッドアプリケーションにおいて，ロックとSTMを共存させ，スレッドが状況に応じてロック，STMを切り替える排他制御機構を提案する．本提案システムではユーザはSTMライクにコードを記述でき，デッドロックなどのロックが持つプログラマビリティの問題を意識することはない．スレッドが情報を収集し，ロック，STMのどちらでクリティカルセクションを実行するかを判断する．STMで極端に性能が低下する状況でロックに切り替えることで，STMよりも高いパフォーマンスが期待できる．

本研究では，オープンソースで提供されているSTMライブラリに提案システムを追加する形での実装を提案した．また，静的に切り替えを行う提案システムのプロトタイプとロック単体，STM単体との性能比較を行い，本提案システムの優位性を示した．

（大学院情報システム学研究科）



平成25年度 修士論文

ロックとSTMを共存させた 排他制御機構の提案

電気通信大学 大学院情報システム学研究科
情報システム基盤学専攻
1253019 村井 栄王

指導教官 多田 好克 教授
小宮 常康 准教授
大森 匡 教授

提出日 平成26年1月27日

目 次

第 1 章 序論 **5**

第 2 章 背景 **7**
2.1 マルチスレッドアプリケーション 7
2.2 従来の排他制御手法 7
2.2.1 ロック 7
2.2.2 ソフトウェアトランザクショナルメモリ 12
2.3 本研究の目的 16

第 3 章 関連研究 **17**
3.1 STM の実装 17
3.1.1 STM の実装 17
3.1.2 ロックベース STM 18
3.1.3 TL2 アルゴリズム 18

第 4 章 設計 **23**
4.1 要求条件 23
4.2 設計概要 23
4.2.1 関数情報リスト 26
4.2.2 スレッド情報リスト 26
4.2.3 TinySTM 27
4.2.4 ユーザーコード 31
4.2.5 スレッドの実行手順 32
4.2.6 STM 側の設計 34
4.2.7 共存の問題 34
4.2.8 ロック側の設計 35
4.3 切り替え基準 36

第 5 章 実装 **38**
5.1 関数情報リスト・スレッド情報リスト 38
5.2 アボート率計算用スレッド 39
5.3 切り替え判断器 40
5.4 ロック処理用の API 42

第 6 章　評価と考察 **44**
6.1　実験環境 44
6.2　予備実験 44
6.3　提案手法ロックの評価 44
6.4　切り替え基準の決定 46
6.4.1　共有メモリ数と性能の関係の検証 51
6.4.2　クリティカルセクションの割合と性能の関係の検証 53
6.5　評価実験 55

第 7 章　結論 **61**

図 目 次

2.1 ロックの粒度の問題 9
2.2 シリアル性の問題の例 10
2.3 ロックでのパフォーマンス低下例 11
2.4 ロックの合成性の問題 12
2.5 STM の動作例 14
2.6 STM のパフォーマンスが低下する例 15

3.1 TL2 の実装イメージ 19

4.1 システムの概要図 24
4.2 TinySTM の概要図 27
4.3 TinySTM でのロック変数の構造 28
4.4 提案システム内でのスレッドの動作フロー 33
4.5 ロックと TL2 STM が競合した場合の例 34
4.6 提案手法ロックのデッドロック例 36

5.1 情報リストのデータ構造 39

6.1 *pthread_mutex_lock* と提案手法ロックの性能比較 45
6.2 高競合時の提案手法ロックと STM の性能比較 50
6.3 クリティカルセクション内で使用する共有メモリ数と性能の関係 . 54
6.4 クリティカルセクションの実行割合が 1%の時の性能比較 56
6.5 クリティカルセクションの実行割合が 5%の時の性能比較 57
6.6 クリティカルセクションの実行割合が 10%の時の性能比較 58
6.7 評価実験 59

表 目 次

4.1 ロックとSTMの状況による性能 25

ソースコード目次

2.1 ロックの擬似コード 8
2.2 デッドロック回避の擬似コード 10
2.3 STM の擬似コード 13
4.1 ロックエントリの算出方法 30
4.2 ロック変数の更新方法 30
4.3 ユーザーコードの例 31
5.1 切り替え判断器のコード 41
5.2 本システムでの排他制御の実行開始例 42
6.1 共有メモリにアクセスするロックのソース 48
6.2 共有メモリにアクセスする STM のソース 48
6.3 休止マクロの定義 49
6.4 共有配列にアクセスするロックのソース 52
6.5 共有配列にアクセスする STM のソース 52

第 1 章

序論

排他制御はマルチスレッドシステムの開発効率上，性能上のボトルネックとなっている．

従来はロックを使用して排他制御を行うのが主流である．この手法は書き込みが多い状況では比較的高速な排他制御を行うことができるが，読み込みが多い状況では比較的遅い．また，ロックの粒度が細かい場合はデッドロックの危険性があり，プログラマビリティが低い．

そのプログラマビリティの問題を解決するために，Touitou らが提案したのがソフトウェアトランザクショナルメモリ（Software Transactional Memory: STM）である．これはデータベースのトランザクションに真似て排他制御を行うものである．STM では，トランザクションは投機的に共有メモリへのアクセスを行い，データの矛盾が発生したことに気づくと排他制御のはじめに戻る（ロールバック）．STM の実装によってはロックを使用しない．また，ロックベース STM の実装ではデッドロックを回避する機能が備わっているため，プログラマはデッドロックを意識することなくアプリケーションを開発することができ，プログラマビリティが高い．また，STM ではリードロックを獲得する必要がなく，コア数だけ共有メモリの読み込みを行うことができるので，読み込みが多いなどのロールバックが起きにくい状況ではロックより高速に動作することができる．しかし，書き込みが多い状況だと逆にロックよりも性能が劣る．

本研究では，マルチスレッドアプリケーションにおいて，ロックと STM を共存させ，スレッドが状況に応じてロック，STM を切り替える排他制御機構を提案する．本提案システムではユーザは STM ライクにコードを記述でき，デッドロックなどのロックが持つプログラマビリティの問題を意識することはない．スレッドが情報を収集し，ロック，STM のどちらでクリティカルセクションを実行するかを判断する．STM で極端に性能が低下する状況でロックに切り替えることで，STM よりも高いパフォーマンスが期待できる．

本研究では，オープンソースで提供されている STM ライブラリに提案システムを追加する形での実装を提案した．また，静的に切り替えを行う提案システムのプロトタイプとロック単体，STM 単体との性能比較を行い，本提案システムの優

位性を示した.

第 2 章

背景

2.1 マルチスレッドアプリケーション

CPU コアの増加により，マルチスレッドシステムの重要性が増している．並列性の高いアプリケーションを構築することで，シングルスレッドで同様の処理をするよりも速く処理することができ，より高いパフォーマンスをユーザに提供することができる．しかしスレッド間で共有メモリの操作を行う場合は，矛盾が発生しないよう，排他制御が必要になる．

排他制御はマルチスレッドアプリケーションにおいて，開発効率上，パフォーマンス上のボトルネックとなる問題である．

2.2 従来の排他制御手法

排他制御の手法はいくつかあるが，本研究ではロックとソフトウェアトランザクショナルメモリ（Software Transactional Memory: STM）に焦点を当てる．

2.2.1 ロック

ロックは従来のマルチスレッドシステムの排他制御に多く使われている手法の 1 つである．これは文字通り共有メモリに対して鍵を掛け，鍵をかけたスレッド以外をブロックする排他制御手法である．一般的なロックを使用した排他制御は，ソースコード 2.1 に示した擬似コードのようなコードを記述することで実現する．

ソースコード 2.1: ロックの擬似コード

```
//共有メモリ
int shred_data;

//クリティカルセクションのスタート
lock( &shared_data );

/*共有メモリへのアクセス*/

//クリティカルセクションの終了
unlock( &shared_data );
```

ロックでは，共有メモリのロックを取得した時点からクリティカルセクションがスタートする．他のスレッドがすでにロックを取得していた場合，ロック獲得処理（5 行目）内で当該ロックが解放されるまで待機（スピン）する．クリティカルセクション内では，共有メモリへのアクセスを行う．最後にロックを解放（10 行目）することでクリティカルセクションが終了となる．

ロックの実装は比較的単純であることから多く使用されている．しかしロックには次のような問題が挙げられる．

- 粒度の問題
- シリアル性の問題
- 合成性の問題

ここで言う粒度とは，ロック 1 つ当たりで管理するデータ量のことを指す．シリアル性の問題とは，変更する可能性が低い共有メモリに対しても 1 つのスレッドしか同時にアクセスすることができないロックのパフォーマンス上の問題を指す．合成性の問題とは，2 つ以上の不可分な処理を合成する時のプログラマビリティの問題を指す．

粒度の問題

図 2.1 は，スレッド間で共有している配列要素 1 つ以上をスレッド A，B が排他制御するプログラムを考えた例である．

この例の場合，配列全体に 1 つのロックを使用する時はロックの粒度は粗い．逆に配列要素 1 つにつき 1 つのロックを使用する場合は，ロックの粒度が細かい．ロックの粒度が粗い場合（図 2.1 左側），ロックは共有メモリ全体で 1 個だけであるため，デッドロックが発生することはない．そのため，ロックの粒度の細かい場合に比べて容易に排他制御システムを開発することができる．しかし，本来並列に動くことが出来るスレッドもシリアルにしか実行できないため，システム全体の

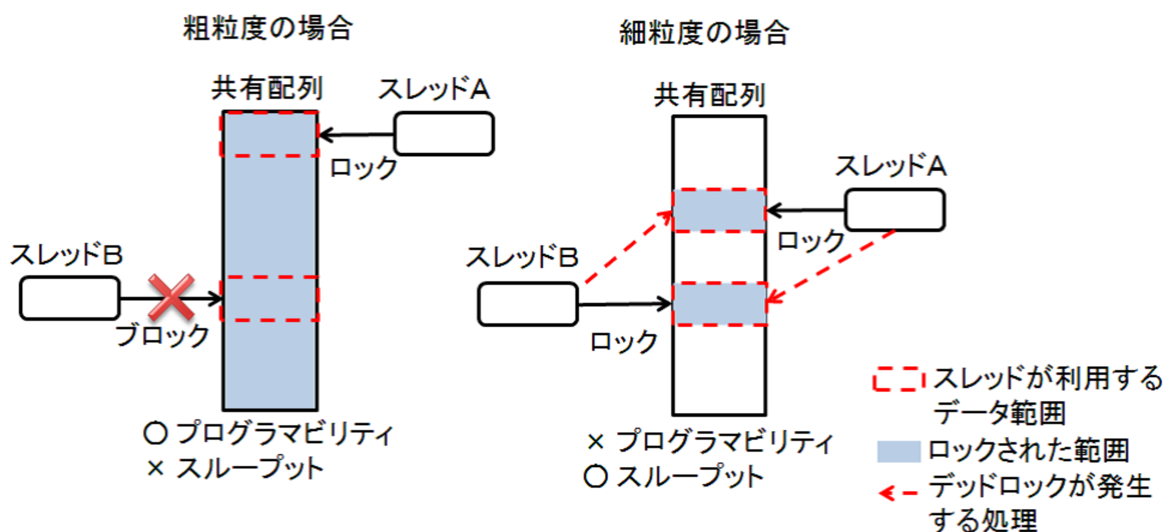


図 2.1: ロックの粒度の問題

性能が低くなる．図 2.1 の粗粒度の場合，スレッド A，B は配列内の異なる場所へアクセスしようとしているため，本来同時に実行することが出来るが，配列全体につき 1 つしかロックがないため，配列へのアクセスはスレッド A，B のどちらか 1 つしか同時に実行することが出来ない．

反対に粒度が細かくなると，スレッド A とスレッド B は並列に動くことが出来るようになる（図 2.1 右側）．そのため，粗粒度の場合よりもプログラムのスループットは高い．しかし，スレッド A，B がお互いに相手がすでにロックを獲得している共有変数のロックを取得しようとした場合，デッドロックが発生する．このように，粒度が細かくなるとプログラムのスループットが高くなると同時にデッドロックの危険性が高くなり，プログラマはデッドロックが発生しないよう注意して開発を行う必要がある．解決方法の 1 つとして，ソースコード 2.2 のように，共有メモリのアドレスの昇順にロックを取得するというルールを作ることによってデッドロックの回避が可能である．

ソースコード 2.2: デッドロック回避の擬似コード

```
//異なる共有メモリ2つの操作をする関数
void critical_section( int *s1, int *s2 )
{
  //ロックを獲得する順番を
  //決定するための変数
  int younger, older;

  //アドレスを比較
  if( &s1 > &s2 ){
    younger = s2;
    older   = s1;
  }else{
    younger  = s1;
    older    = s2;
  }

  //クリティカルセクションの開始
  //昇順にロックを取得
  lock( younger );
  lock( older );

  /*排他制御処理*/

  //クリティカルセクションの終了
  unlock( older );
  unlock( younger );
}
```

ソースコード 2.2 では，ローカル変数 younger，older を使用して，アドレスの数値が小さい方を younger，大きい方を older に格納し，昇順にロックを獲得している．このように，細粒度でロックのプログラムを記述する場合はデッドロックが発生しないように注意が必要である．

以上のことから，ロックは粒度が粗いと性能が低下し，粒度が細かいとプログラマビリティが低下する．

シリアル性の問題

図 2.2 はスレッド A，B，C が同じ共有メモリに対してロックを使用してアクセスする例である．

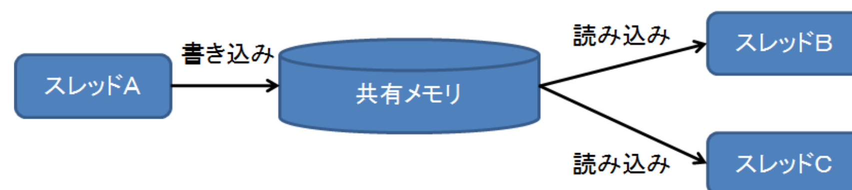


図 2.2: シリアル性の問題の例

スレッド A だけが共有メモリに対して書き込みを行い，スレッド B，C は値を読み込むだけだとする．スレッド B，C が同時に実行を始め，その後にスレッド A が実行を始める状況を考える．ロックは図 2.3 左側のようなフローで排他制御が実行される．

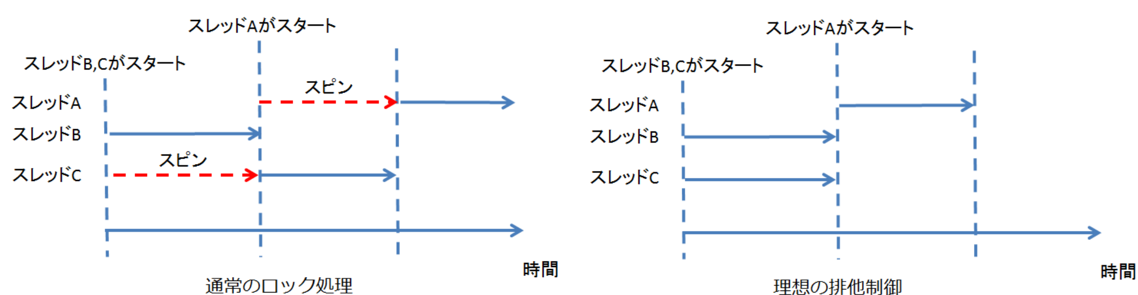


図 2.3: ロックでのパフォーマンス低下例

図 2.3 左側ではスレッド B が先にロックを取得し，スレッド A はスレッド B の操作が終了するまで待機する．しかし，この場合はスレッド B，C は共有メモリに対して読み込みを行うだけである．共有メモリを変更しないので，このような場合に限って同時に実行しても共有メモリの一貫性を保つことができ，性能が向上する．しかし，ロックの場合はシリアルに実行をしなければデータの一貫性を保証することが出来ないので，ロックのセマンティクス上理想の排他制御の動作をロックで行うのは不可能である．

スレッド B，スレッド C は同時に実行することが出来ない．本研究では，このような問題をシリアル性の問題と呼ぶ．

合成性の問題

また，ロックのプログラマビリティの問題として合成性の問題がある [6]．共有メモリである 2 つのキューを操作するアプリケーションの例を図 2.4 に示す．

キューからデータをデキューする処理と，キューの末尾にエンキューする処理があるとする．これらの処理はどちらもアトミックに行うことが出来るとする．つまり，操作の前にキューをロックすることで，操作の不可分性を保証する．ここに片方のキューからデキューしたデータを，もう片方のキューにエンキューするという共有メモリの操作を追加しようとすると問題が発生する．

単純にデキュー処理とエンキュー処理を組み合わせるだけでは，この処理はアトミックに実行することは出来ない．スレッド A がキュー A からデキューし，キュー

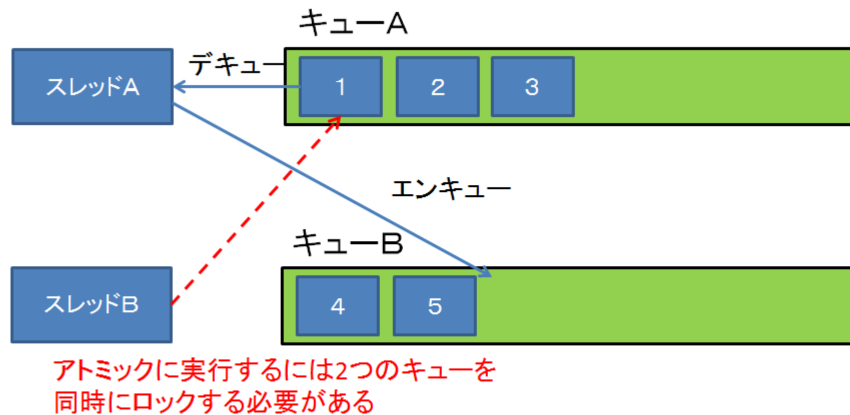


図 2.4: ロックの合成性の問題

B にエンキューする場合を考える．単純にデキュー処理とエンキュー処理を組み合わせた場合，スレッド A はキュー A をロックを獲得し，データ 1 をスレッドローカルに一時的に保存し，キュー A のロックを解放する．そして，キュー B のロックを獲得し，データ 1 をエンキューしてからキュー B のロックを解放するという流れになる．これでは，2 つの処理をアトミックに行うことが出来ていない．つまり，スレッド A がキュー A からデキューした直後にスレッド B がキュー A にアクセスした時，データ 1 はキュー B の末尾になければならないが，単純にデキュー処理，エンキュー処理を合成しただけではこの 2 つの処理をアトミックに行うことは出来ない．

解決方法の 1 つとして，ロックの粒度をキュー A，キュー B の 2 つで 1 つのロックを使用することが考えられるが，粗粒度になることでパフォーマンスが低下する（2.2.1 節）．したがって，プログラマはデキュー，エンキューのコードを再利用することはできず，新規に処理を記述する必要がある．

2.2.2 ソフトウェアトランザクショナルメモリ

ロックのプログラマビリティの問題を解決するため，Touitou らが初めて提案した排他制御手法がソフトウェアトランザクショナルメモリ (Software Transactional Memory: STM)[1] である．この手法は，データベースのトランザクションに動作を真似て排他制御を行う手法である．STM では，ACID(Atomicity, Consistency,

Isolation, durability）特性のアトミック性，一貫性，独立性を保障するものになっている．STM にはいくつか異なる実装が提案されている．本節では，STM の概要だけ説明し，提案されている STM のアルゴリズムの説明については 3.1 節で述べる．

STM の概要

STM では，スレッドが排他制御を行うコード領域を「トランザクション」としてマークし，スレッドはマークされた領域を投機的に実行する（ソースコード 2.3）．

ソースコード 2.3: STM の擬似コード

```
1  //共有メモリ
2  int shared_data;
3
4  //ローカル変数
5  int local;
6
7  //トランザクションの開始
8  TM_START;
9
10 //共有メモリの読み込み
11 local = TM_LOAD( &shared_data );
12
13 //共有メモリへの書き込み
14 TM_STORE( &shared_data, local );
15 //トランザクションの終了
16 TM_COMMIT
```

ロック（ソースコード 2.1）では lock 関数でクリティカルセクションが始まり，unlock 関数でクリティカルセクションが終了していたのに対して，STM では *TM_START* でクリティカルセクションが始まり，*TM_COMMIT* でクリティカルセクションが終了する．この範囲がトランザクションと呼ばれるコード領域である．STM ではこのコード領域で行う共有メモリへの操作の排他性を保証する．

トランザクションを実行中に共有メモリの一貫性が保たれていないことが分かった場合，競合マネージャがどのトランザクションをアボートするべきかを決定し，アボートするべきと判断されたトランザクションを持つスレッドは現在のトランザクションの処理を放棄し，トランザクションの最初の部分までロールバックを行う．

図 2.5 は，スレッド A，B がトランザクション実行中に競合を起こした場合の例である．スレッド A，B ともに同じ共有メモリに対してアクセスをしているとする．スレッド B のトランザクションの書き込みが先に終了したため，スレッド A が読み込んだ共有メモリの値と矛盾が生じる．この場合，スレッド A は現在の処理を放棄し，ロールバックを行う．このように動作することで，STM は共有メモリの一貫性を保証する．

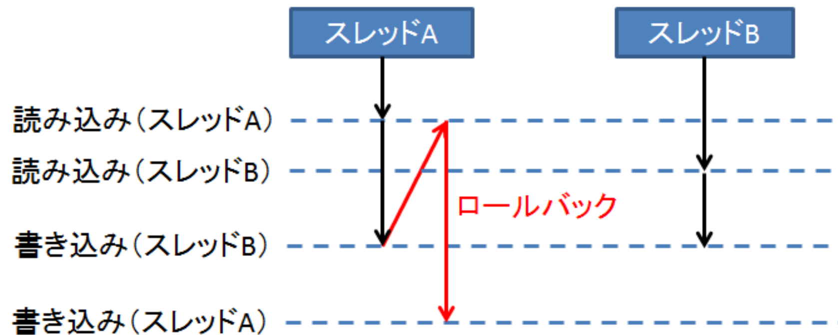


図 2.5: STM の動作例

STM はトランザクションが共有メモリの矛盾を認識した時，クリティカルセクションの始めにロールバックをすれば良いという楽観的な考えを元に動作する排他制御手法である．それに対してロックはクリティカルセクションの始めに共有メモリのロックを獲得して排他制御を行う，いわば悲観的な排他制御手法である．この楽観的なセマンティクスにより，ユーザはデッドロックを意識する必要なくコードを記述することができる．そのため，ロックと比較してプログラマビリティが高い．

また STM は，同じ共有メモリに対して読み込みを行う複数のスレッドが同時に実行されている（2.2.1 節で説明した状況）場合，ロックを行わないため同時に実行可能である．したがって，図 2.2 右側のようなフローで実行可能というパフォーマンス上の利点もある．

そして，トランザクションとしてマークした範囲の処理は結果的に不可分に行われていることが保証されているので，複数のアトミックな処理を容易に合成することが出来る．

しかし，STM には次のパフォーマンス上の問題点がある．

- メタデータ操作によるオーバヘッド
- ロールバック処理によるオーバヘッド

実装によって差異があるが，共有データの一貫性を保つため，STM は基本的にスレッドローカルに共有メモリのメタデータをもつ必要がある．メタデータに共有メモリの場所と値を保存しておき，コミットの時に共有メモリの値とスレッド

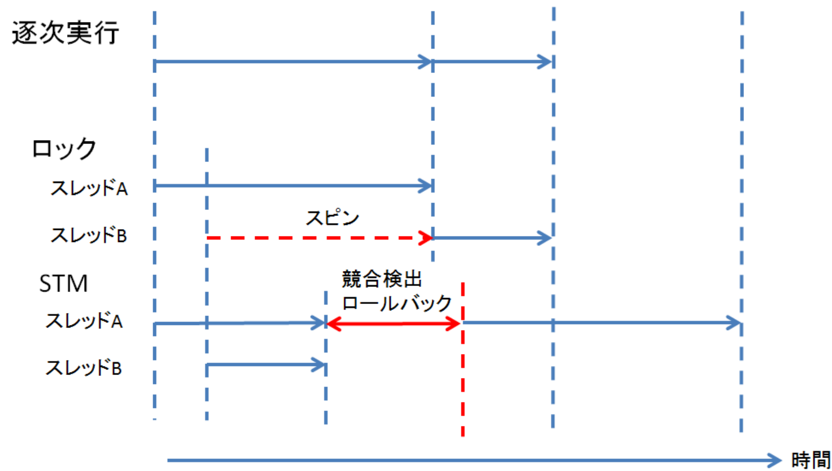


図 2.6: STM のパフォーマンスが低下する例

ローカルのメタデータとの比較を行うことで，一貫性を保つ．このように，ロックにはないオーバヘッドが生じるので基本的にロックより低速である [3].

また，STM は共有メモリの一貫性が保たれなかった場合に行うロールバックのため，書き込みの競合が発生した場合にスループットが低下する．図 2.6 は，同じ共有メモリに対しての書き込み競合によって STM のスループットが低下する例である．

図 2.6 では，ロックはスレッド B が先にロックを獲得したスレッド A の排他制御が終了するまでスピンをする．スレッド B はスレッド A の排他制御が終了した直後にクリティカルセクションに入ることが出来るので，ロック処理が十分に短いとすれば逐次実行と同じだけの実行時間で終了する．しかし STM では，スレッド B は，先に排他制御を始めているスレッド A を認識することが出来ないので排他制御を開始し，スレッド B の排他制御の方が短いために先に終了する．スレッド A は共有メモリの矛盾に気がつきロールバック処理を行う．このロールバック処理の時間だけ逐次実行よりもスループットが低下する．したがって，書き込み競合が頻発することが予想されるアプリケーションではスループットが低下する可能性が高い．

2.3 本研究の目的

本研究では，マルチスレッドスレッドアプリケーションにおいて個々のスレッドがロック，STM に動的に切り替え，混在して機能する排他制御機構を提案する．クリティカルセクションを実行する最初の段階，もしくはトランザクションのロールバックによってクリティカルセクションの頭の部分まで戻ってきた際に情報を収集し，これから行う排他制御処理はロック，STM のどちらで行うのが有利であるのかをスレッド自身が判断をする．

本提案手法では，ユーザーは STM ライクに排他制御のコードを記述することができ，複雑な排他制御処理を記述する際もデッドロックを意識する必要がない．また，STM で性能が低下する状況でロックに切り替えることで，STM 単体で実行するよりも高いパフォーマンスが期待できる．

本研究では，次のことが課題となる．

- ロック，トランザクションが混在しての正常な排他制御
- どのような状況でロック，STM に切り替えるべきか

本研究では，スレッドがロック，STM のどちらでクリティカルセクションを実行するかを決定する．したがって，同じクリティカルセクションでもロックを実行中のスレッドと STM を実行しているスレッドが同時に存在する可能性がある．いくつか提案された STM 実装の中で，どの実装がロックとの共存に適しているか，また，どのような切り替え基準を設けるのが本提案手法で最適なのかを調査する必要がある．

本研究では，TL2 アルゴリズム [5] とロックの共存の手法を検討し，独自のロックライブラリを作成した．また，ロック/STM の性能が上がるであろう状況を作り出し，予備実験として STM の TL2 アルゴリズムと作成したロックとの性能比較を行った．それを元に，手動で切り替えを行う提案システムのプロトタイプとの性能比較を行い，本提案手法の優位性を示した．

第 3 章

関連研究

3.1 STM の実装

本節では，今までに提案されてきたいくつかの STM の実装について説明する．

3.1.1 STM の実装

STM には，実装に関して以下に示すような選択肢がある [4].

- オブジェクトベース or ワードベース
- ライトスルー or ライトバッファ

オブジェクトベース STM は，ロード，ストアをオブジェクトベースで行う STM の実装である．1 度のロード，ストアで多くのデータをやり取りできるが，この実装にはオブジェクト指向言語のサポートが必要である．対して，ワードベース STM は，実行環境の 1 ワードでトランザクションのロード，ストアを行う実装である．この実装はオブジェクトベースのように言語サポートが必要ない代わりに，1 度のロード，ストアに 1 ワードだけをやり取りする実装である．したがって，アーキテクチャによって 1 度にやり取りできる容量が異なることと，1 度に 1 ワードだけでしかやり取りができないので，容量の大きい共有データ構造をやり取りする場合は 1 ワードのメタデータを使用して間接化するなどの工夫が必要になる．

ライトスルー方式は，共有メモリの値をバックアップ用領域（アンドゥログ）に保存しておき，共有メモリの場所に直接トランザクション内の変更を書き込む方式である．この方法は競合の検出を積極的（Eager Conflict Detection）に行うことができるが，競合検出時にアンドゥログから元の値を読み込み，書き戻すオーバーヘッドがかかる．それに対してライトバッファ方式は，トランザクションが行った変更をスレッドローカルに一時的にバッファしておく方式である．ローカルに変更を溜めておくため，共有メモリを元の値に書き戻すオーバヘッドがかからないが，競合検出が遅れる（Lazy Confilict Detection）ことになる．

3.1.2 ロックベース STM

STM が Touitou らによって提案された [1] 時は，ノンブロッキングな排他制御方式であった．つまり，コンペアアンドスワップなどの命令を使用して，アトミックに共有メモリの値を書き換える手法をとっていた．しかし，この手法は複雑なアルゴリズムであり，オーバヘッドが大きい．また，多くのメタデータを扱う必要があるために処理効率が悪かった．

この問題を解決するため，Ennals によって初めて提案されたのがロックベース STM[8] である．ロックベース STM は，STM のある 1 部分についてだけロックを使用する実装である．ロックを使用することで，シンプルなアルゴリズムとなり，オーバヘッドが小さくなる．また，スレッドが持つメタデータが少なくなったことでキャッシュに多くのデータが乗るようになり，高速になるという利点がある．ロックベース以前の STM はメタデータを多く使うため，組込みシステムのようなコンピュータ資源の乏しい環境には適さなかったが，ロックベース STM を使用して組込み環境上でどのような実装が最も効率的に動作するかという研究がある [10].

ロックを使用することで，デッドロックが発生するリスクが発生することになるが，一般的なデータベースのようにタイムアウトを設定し，一定時間以上スピンしたトランザクションはアボートする．また，STM 専用にデッドロック検出を行い，タイムアウト方式より高効率にトランザクションを実行する研究がある [7].

3.1.3 TL2 アルゴリズム

ロックベース STM で主流となっているのが，Dice らが提案した TL2 アルゴリズム [5] である．この実装はワードベースのライトバッファ方式で排他制御を行う STM である．TL2 アルゴリズムは，図 3.1 のような実装イメージとなる．

TL2 の実装では，スレッドはリードセット，ライトセットと呼ばれる共有メモリのメタデータを持っている．また，リードバージョン（rv），ライトバージョン（wv）と呼ばれるデータを持っている．リードセットはトランザクションが共有メモリをロードした時に共有メモリのアドレス，値，ロードした時の共有メモリのバージョン番号を保存するメタデータのリストである．ライトセットは，ロードした共有メモリの値を変更した時に，リードセット内のメタデータのアドレス，新しい値を保存するメタデータである．リードバージョンはトランザクションを開始した時のグローバルバージョンクロックの値を保存しておくためのデータであり，ライトバージョンはコミットする時のグローバルバージョンクロックの値を保存するデータである．

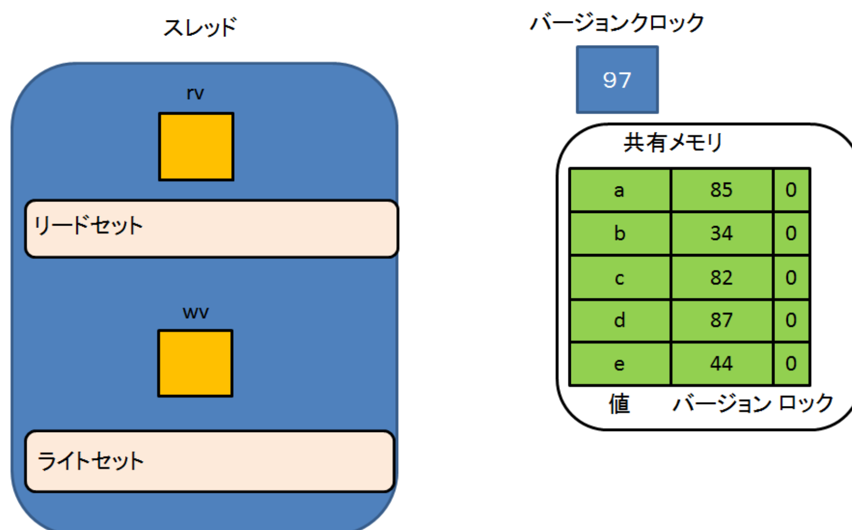


図 3.1: TL2 の実装イメージ

TL2 アルゴリズムの特徴は次のようになる．

- グローバルバージョンクロックを使用する
- コミット時にだけロックを使用する
- ロックとバージョン番号を使用して競合を検出する

グローバルバージョンクロックは，共有メモリの一貫性を保つための擬似的な時間を示すデータ構造である．トランザクションはスタート時にグローバルバージョンクロックの値をローカル変数 rv に保存し，コミットに成功する時にグローバルバージョンクロックの値をインクリメントする．また，共有メモリには最後にコミットに成功したグローバルバージョンクロックの値とロックビットが関連付けられている．

TL2 アルゴリズムでは，コミットを行う時にロックを使用する．詳細は後述するが，書き込みを行う場所にだけロックを獲得する．

トランザクションが読み込みを行う時に，共有メモリがロックされていた場合，そのトランザクションは一貫性を保つことが出来ないのでアボートする．また，トランザクションのはじめに取得したグローバルバージョンクロックの値よりも大きい値が共有メモリに対応したバージョン番号に格納されていた場合も同様の理由でアボートする．

書き込みトランザクションの動作　次の一連の操作は書き込みトランザクションによって実行されるものである．

1. グローバルバージョンクロックの取得
2. 投機的な実行の開始
3. ライトセットのロック
4. グローバルバージョンクロックのインクリメント
5. リードセットの検証
6. コミットとロックの解放

トランザクションはまず，グローバルバージョンクロックの値を読み込み，スレッドローカルの変数 rv に保存する．この値はライトセットのロックをした時に共有メモリのバージョンと比較するのに使用する．

投機的な実行では，トランザクションコード（ロードやストアなどの命令は投機的な実行が共有メモリの状態をコミット時まで変更しないように機械的に拡張，置き換えられる）を実行する．共有メモリをトランザクションがロードする時，まずは読み込むアドレスがすでにライトセットにないかチェックをする．もしライトセットにあるようであれば，トランザクションのロードはそのアドレスに書かれた値を返す．ライトセットになければ，共有メモリのアドレスと値，共有メモリのバージョンをリードセットに保存する．トランザクションが共有メモリに対してストアする時は，共有メモリには書き込まずに対応したライトセットのエントリに新しい値を書き込む．

なお，トランザクションがロード，ストアする時には，共有メモリのロックビットとバージョンを確認する．ロックビットが 1 であった場合，他のスレッドがその共有メモリに対してコミットを行うため，アボートをする．また，共有メモリのバージョンがトランザクションのリードバージョン（rv）より大きい場合も，一貫性が保てない可能性があるのでアボートする．

投機的な実行が終わると，次は任意の順序でライトセット内のすべての対応した変数のロック獲得を行う．また，ロックの獲得待機時間を有限にする．これはデッドロックを回避するためである．これらのロックがすべて正しく獲得できなかった場合，トランザクションは失敗となり，アボートする．

ライトセット内のすべてのロックの獲得に成功すると，ローカル変数 wv にグローバルバージョンクロックに対してアトミックなインクリメントアンドフェッチ操作を行って得た値を書き込む．

ロック獲得後，リードセット内の各エントリに対応した共有メモリのバージョンの検証を行う．バージョン管理されたライトロックに関連付けられたバージョン番号は rv 以下であるかどうかを確認する．これらのメモリ位置が他スレッドによってロックされていないかも同時に確認する．検証に失敗した場合，トランザクションはアボートする．リードセットをコミット段階で再検証することで，投機的な実行を行っている間に共有メモリが変更を加えられていないことを保証する．

例外として rv + 1 = wv となる特別なケースの場合，リードセットを検証する必要はない．並行に実行しているトランザクションがこの値を変更した可能性がないためである．

検証が終わると，ライトセット内の各位置に対して，ライトセットからの新しい値をストアし，wv の値を対応したバージョンフィールドに書き込み，ライトロックビットをクリアすることでロックを解放する．

読み込みトランザクションの動作　読み込み専用トランザクションは次のように動作する．

1. バージョンクロックの値を獲得

2. 投機的な実行を行う

バージョンクロックの値の獲得は書き込みトランザクションのはじめの動作と同じである．投機的な実行は，共有メモリのロックビットがフリーであることと，ロックのバージョンが rv 以下であることを確かめたあとにロードが実行される．もし rv よりも大きい場合は，コミットを行わずにアボートする．

読み込み専用のトランザクションは上記のとおり処理が軽量になるため，高効率にトランザクション処理を行うことが出来るという利点がある．

ロックをコミット時にだけ使用することと，読み込みトランザクションが単純になることで，共有メモリに対して読み込みトランザクションが多い状況では他の STM の実装より高い効率で排他制御を行うことが出来る．しかし，検証を読み込み時とコミット時にだけ行うので，検証が怠惰 (Lazy conflict detection) なものになってしまう．

以降，本研究では TL2 アルゴリズムに準拠したものを TinySTM[2] 上で実装し，実験，測定等を行う．

第 4 章

設計

4.1 要求条件

本研究では，次のことを前提条件として設定する．

- ユーザは STM ライクにコードを記述することができる
- ロックコードはコンパイル時に生成する
- ロック，STM のコードを状況に応じて動的に切り替える

ユーザは本システムを使用することで，STM ライクに排他制御コードを記述することができる．そのため，ロックでは意識しなければならなかったデッドロックと合成性を意識する必要がなくなる．

ユーザが記述した排他制御処理にロックのコードは当然含まれていないので，システム側でロックのコードをコンパイル時に生成する．つまり，同じ排他制御の STM 版の実行コードとロック版の実行コードの 2 種類が存在することになる．また，このような実装にすることによる条件として，本提案システムで扱う共有メモリは位置が静的なものに限定する．つまり，排他制御を行う場所が実行中に動的に変更するようなものは本システムでは対応しない．

スレッドは基本的に動的に STM で排他制御を行う．クリティカルセクション実行時もしくはトランザクションがアボートした時にスレッドが情報の収集と切り替えの判断を行い，ロックで実行する方が良いと判断した場合は，当該クリティカルセクションを終了するまでロックで動作する．学習機能のようなものは考慮していないので，同じスレッドが過去にロックを選択したクリティカルセクションを再び実行する時には，最初にクリティカルセクションを実行した時と同様に排他制御の初めに STM，ロックのどちらで実行するべきかを判断する．

4.2 設計概要

本研究での位置づけは図 4.1 のようになる．

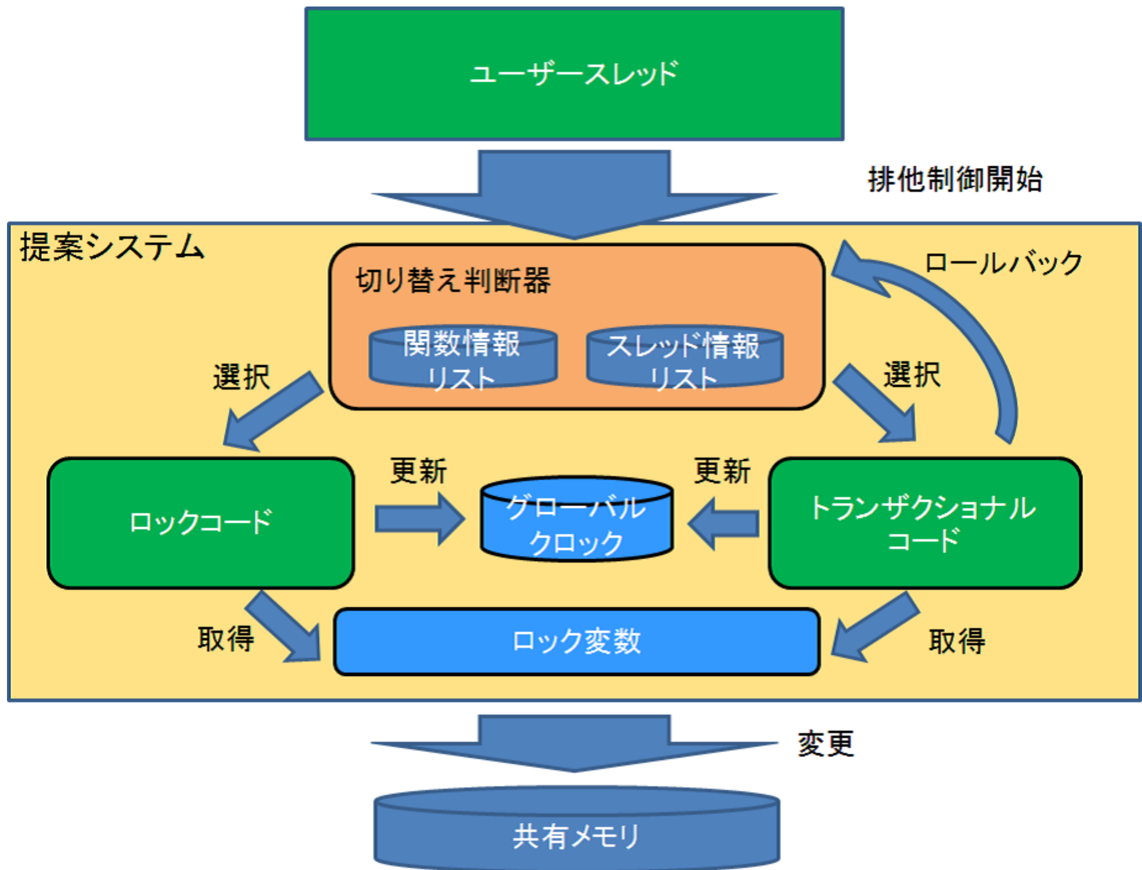


図 4.1: システムの概要図

後述するが，STM は TL2 の実装を使用する．ユーザによって記述された排他制御コードは，コンパイル時にロック，STM の実行コードを生成する (図 4.1 内「ロックコード」,「STM コード」部)．ユーザースレッドはクリティカルセクション実行開始時にロック，STM のどちらで実行するかを切り替え判断器を使用して判断する．

いずれのコードもグローバルバージョンクロックと，ロック変数を使用して排他制御を行う．具体的な使用方法については，4.2.6 節と 4.2.8 節で述べる．

また，本研究では一般的なロックと STM の性能は，相対的に表 4.1 のようなものになると考えている．

表 4.1: ロックと STM の状況による性能

	ロック	STM
読み込み競合が多い	×	○
書き込み競合が多い	○	×
使用する共有メモリ数が多い	○	×
使用する共有メモリ数が少ない	△	○
クリティカルセクション内の計算量が多い	○	△
クリティカルセクション内の計算量が少ない	△	○

読み込み競合が多い場合，ロックの場合はリードロックを獲得する必要があるため，シリアルにしか実行が出来ない．それに対して STM はリードロックを獲得する必要がないため同時に実行をすることが出来る．しかし，書き込み競合が多い状況では STM はロールバックが頻発するため，ロックでシリアル実行する方が高速になる可能性が高い．同じクリティカルセクション内で多くの共有メモリを使用するだけトランザクションの競合によるアボートするリスクが高まるため，ロックの方が高速である可能性が高い．また逆の場合は，読み込み競合が多い状況であれば STM が高速に動作する可能性が高い．クリティカルセクション内の計算量などが多い場合は，トランザクションがアボートする確率が高くなる．つまり，共有メモリの値を使用して複雑な計算処理をする場合や，ネットワークを通じて排他制御をするシステム [9] のような，クリティカルセクション内の実行時間が長くならざるを得ない排他制御の場合，STM では同時に排他制御を実行しているスレッドが多いほどアボートする可能性が高くなる．したがってクリティカルセクション内の計算量が多い場合はロック，少ない場合は STM が有利である．

情報の収集には，提案システム内の関数情報リスト，スレッド情報リストを利用する．

4.2.1 関数情報リスト

関数情報リストは，クリティカルセクションを含んだ関数の情報をまとめたリストで，次の情報を格納する．

- 関数の場所（ポインタ）
- 実行中のスレッド数
- 読み込み専用か否か

関数はすべての関数が登録されるわけではなく，排他制御処理を含んだ関数に限定される．リスト内で関数を識別するため，関数の場所を記録する．切り替え判断処理において，競合状態の参考にするために実行中のスレッド数を記録する．スレッドは排他制御開始時に当該関数の情報ノードのスレッド数部をインクリメントし，排他制御終了時にデクリメントする．関数内で行う排他制御処理が読み込み専用である場合，STM ではリードビフォーライト，もしくはリードアフターライト競合が発生しない限りは並列に動作を行うことが出来るので，ロックよりも高速に処理を行うことが出来る可能性が高い．

4.2.2 スレッド情報リスト

スレッド情報リストには，それぞれ次の情報が格納される．

- スレッド ID
- 実行中の関数ポインタ
- 排他制御開始時間
- 累積のアボート回数
- 単位時間あたりのアボート回数

スレッド ID は，スレッドが情報収集をする時にリストから自分の情報ノードを探すための識別子になっている．排他制御開始時，実行中関数ポインタ部分にこれから実行する関数のポインタを登録する．トランザクションがアボートした時，再びスレッド情報リストを参照し，累積アボート数をインクリメントし，切り替え判断処理をもう 1 度行う．排他制御を開始した時間を記録しておき，アボート率と経過時間が一定以上経過している場合はロックに切り替える．単位時間当たり

のアボート率は，ユーザが作成したスレッドからは参照するだけで，計算，書き込みは行わない．スレッド情報リストのアボート率を計算する専用のスレッドがシステムの初期化の時に実行され，累積のアボート回数を参照し，スレッドローカルに持っている前回に計算した時間を元にアボート率を計算する．計算した結果をアトミックに書き込む．直後に，累積のアボート数を 0 にアトミックにクリアする．

4.2.3 TinySTM

本研究では，TL2 アルゴリズムで実行する STM に TinySTM[2] を使用する．TinySTM はオープンソースなワードベース STM ライブラリである．TinySTM では *uintptr_t* 型のデータを 1 ワードとして扱う．つまり，32bit 環境の場合は 1 ワードが 32bit，64bit 環境の場合は 64bit が TinySTM で 1 度にロード，ストアできる最大のデータ量である．

TinySTM の概要を図 4.2 に示す．

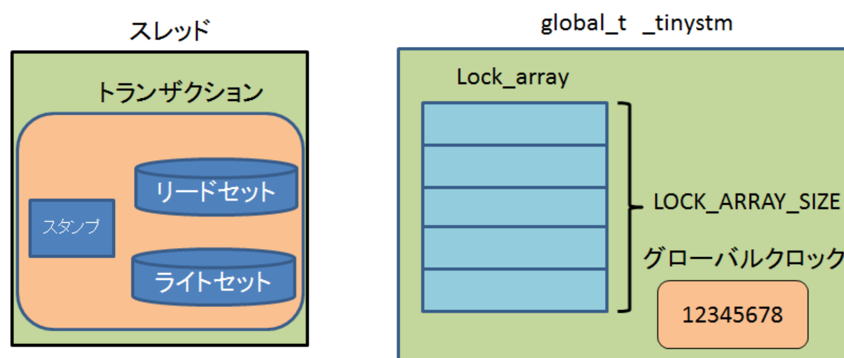


図 4.2: TinySTM の概要図

スレッドは排他制御を行う時，スレッドはローカルに *stm_tx_t* 型構造体を持つ．その中には主に開始時のグローバルバージョンクロックの値を格納するスタンプと，リードセット，ライトセットがある． TinySTM 内のシステム内にグローバルなデータ構造として *_tinystm* が用意されている．この中には，グローバルバージョンクロックと，ロック変数の配列が用意されている．TinySTM では，ロック変数配列は *LOCK_ARRAY_SIZE* というマクロで決定されている．デフォルトは

2^{20} のエントリが用意されている．TinySTM でのロックの粒度は 32 ワードとなっている．

また，32bit 環境の場合，ロック変数は次のようなデータ構造を持っている．

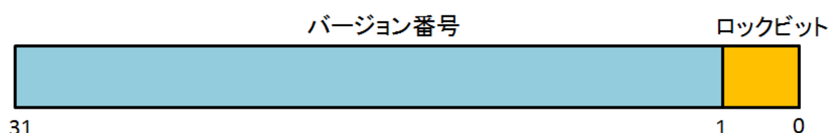


図 4.3: TinySTM でのロック変数の構造

最下位ビットがロックビットとなっており，1 ならロック，0 ならフリーとなっている．それ以外の領域はバージョン番号となっている．これは，トランザクションがコミットに成功した際，ロックビットの解放と同時に最新のグローバルバージョンクロックの値を書き込むための領域になっている．ロック変数も *uintptr_t* 型のデータになっており，32bit 環境と 64bit 環境ではデータ量が異なるが，64bit 環境の場合はバージョン番号領域が拡張されているだけである．

ライブラリファイル (libstm.a) 生成時に，Makefile 内のパラメータを変更することで数種類の STM の動作を選択することができる．本研究で使用する時点での TinySTM のバージョン（1.0.4）では，次の大まかな STM の動作を指定することができる．

- Write Back Encounter Time Locking
- Write Back Commit Time Locking

以上の 2 つの動作は，それぞれどのタイミングでロックを取得するかということを決める．Write Back Encounter Time Locking は，Ennals が提案した STM を行うロック方式である [8]．Ennals の STM では，トランザクションが共有メモリの競合を検出した時にロックを獲得してロールバックする方式である．この方式により，競合を早期に発見することができ，ロールバック後に確実に当該共有メモリの更新を行うことが出来る．Write Back Commit Time Locking[5] は，Dice らが提案したロック方式である．このロック方式については，3.1.3 節で説明をしているので省略する．

また，それとは別に次の細かい動作を指定することができる．

- ライトスルー方式 or ライトバッファ方式

- 競合マネージャの設定

ライトスルー，ライトバッファ方式については，3.1 節にて説明をしたので省略する．TinySTM では，競合を検出したトランザクションがどのような振る舞いをするかを競合マネージャの設定で指定できる．

- 即座にアボートする
- 獲得したロックを保持したままアボート
- 即座にアボートしたのちにランダム時間スリープ

即座にアボートする方法は，競合を検出したスレッドが即座にアボートし，ロールバックする方法である．競合マネージャの設定の中で一番シンプルな方法となっている．獲得したロックを保持したままアボートする方法は，（TL2 の場合）トランザクションがコミット段階まで進んだが，共有メモリの中の 1 つ以上がロックまたはバージョンの検証の時に競合を検出した場合に効果的な方法である．

3.1.3 節に一番近い形で TinySTM を動作させるには，Makefile のパラメータを次のように設定する．

- Write Back Commit Time Locking
- ライトバッファ方式
- 即座にアボートする競合マネージャ

TinySTM での TL2 の動作　TinySTM での TL2 アルゴリズムは，次のようにトランザクションが動作をする．

1. グローバルバージョンクロックの値の取得
2. 投機的なロード，ストアの実行
3. ロックの獲得
4. フェッチアンドアッドによるアトミックなグローバルバージョンクロックの更新
5. 共有メモリの更新
6. ロックの解放

トランザクション開始時に図 4.2 内のグローバル変数 *_tinystm* 内のグローバルバージョンクロックの値を取得する．その後，ライトバッファ方式でリードセット，ライトセットに共有メモリの値や変更した共有メモリの値を書き込む．この時，3.1.3 節と異なる点は，トランザクションが読み書きを行うたびに共有メモリのロック変数を確認することである．したがって，Dice らが提案した手法よりも早期に競合を検出することが出来るようになっている．トランザクションがコミットの段階に入ると，TL2 アルゴリズムと同様にライトセット内に保存された共有メモリに対応したロック変数のロックを行う．

TinySTM では，たとえば共有変数 A に対応したロック変数のエントリを次のように計算する．

ソースコード 4.1: ロックエントリの算出方法

```
1 lock_entry = _tinystm.lock + ( ( &A >> 5 ) & ( LOCK_ARRAY_SIZE - 1 ) );
```

_tinystm.lock は，TinySTM のロック変数配列である．共有変数のアドレス 6 ビット目以上の数値から，ロック変数配列数 –1 の値をマスクとして使用し，配列の添え字を決定している．この算出方法から分かるとおり，TinySTM では 1 ロックで隣接した $2^5 = 32$ エントリを管理することに注意が必要である．本提案手法のロックでこのロック変数を使用することになるので，細粒度ロックではなく，32 エントリ分粒度が粗くなる．

スレッドはコミット時，トランザクション内で一時変更を加えた共有メモリに対応したロック変数に対して，ソースコード 4.2 のような計算処理をすることでロックする．

ソースコード 4.2: ロック変数の更新方法

```
1 Compare_And_Swap( lock_entry, リード時に読んだ値, リード時に読んだ値 | 1 );
```

変数 *lock_entry* の算出方法はソースコード 4.1 と同じである．リード時にリードセットに保存した値（ソースコード 4.1 内「リード時に読んだ値」部）とロック変数の値が変わっていないか検証し，値に変更がなかった場合は最下位ビットを 1 にした値を *lock_entry* 部に書き込む．

TinySTM では，ライトセット内の変更を加えた共有メモリに対応したすべてのロック変数を獲得した時，グローバルバージョンクロックの値をフェッチアンドアッドでアトミックにグローバルバージョンクロックの値をローカルにフェッチし，インクリメントする．

また，リードセット内の共有メモリのバージョンの検証を行い，コミット可能であれば共有メモリの更新を行う．

4.2.4 ユーザーコード

本提案システムを利用するプログラマは，STM ライクにコードを書くと同時に排他制御処理の登録を行う処理をメイン関数のはじめに記述する．これは本提案システム内でスレッドがロックまたはSTMのどちらで排他制御を行うかを判断する材料にするためである．将来的にはこの部分はコンパイル時に解析を行うなどして自動化が期待できる部分であるが，現段階ではユーザに手動で入力をするものとして設計している．

ソースコード 4.3 は，ユーザーがクリティカルセクションを持つ関数を関数リストに登録を行う例である．

ソースコード 4.3: ユーザーコードの例

```
//読み込みだけ，読み書きを行うことを示すパラメータ
#define RO  0
#define RW  1
//スレッド変数
thread_t thread;
//排他制御を含んだ関数
//この関数は読み込みだけを行うものとする
void *critical_section( void )
{
  //クリティカルセクションのスタート
  hymutex_start();

  /*共有メモリへのアクセス*/

  //クリティカルセクションの終了
  hymutex_commit();
}

//メイン関数
int main( void )
{
  //提案手法の初期化
  hymutex_init();
  //関数の登録
  register_func( critical_section, RO );

  //スレッドの開始
  thread_create( &thread, critical_section );

  //スレッドの終了まで待機
  thread_join( thread );
  //スレッドの終了処理
  hymutex_exit();

  return 0;
}
```

関数情報リストにクリティカルセクションを含んだ関数を登録する関数として，ユーザは *register_func* 関数を利用する．これは，第一引数に関数ポインタ，第二引数に読み込み専用か否かを示す数値を代入する．ソースコード 4.3 では，クリ

ティカルセクションを含む関数 *critical_section* は読み込み専用であるので，第二引数にはマクロ RO を代入する．

関数情報リストは，ユーザーが main 関数内で手動で次の情報を登録する．

- 関数のポインタ
- クリティカルセクションは読み込み専用か否か

関数のポインタはクリティカルセクションを持つ関数のポインタである．切り替え判断を行う時に，このポインタを参照して情報を獲得する．

クリティカルセクションが読み込み専用である場合，トランザクションはリードロックをかける必要がないため，パラレルに動作することができる．ただし，クリティカルセクション内の処理量が非常に多い場合はロールバックが発生し，ロックより性能が劣る場合がある．これに関しては後述するスレッド情報リスト内の情報を使用して解決する．

4.2.5 スレッドの実行手順

本提案システムでは，スレッドは図 4.4 のような動作を経て排他制御を行う．

スレッドは排他制御開始前に切り替え判断処理を行い，ステータスを取得する．ステータスとは，これから行う排他制御をロック，STM のどちらの手法で実行するかというパラメータである．当該クリティカルセクションで初めて切り替え判断処理に入った場合，関数情報リスト内のメンバである「実行中スレッド数」をアトミックにインクリメントする．ステータス取得処理では，スレッド情報リスト，関数情報リストからこれから行うクリティカルセクションの情報を取得し，STM，ロックどちらで実行するのが有利かをスレッド自身が判断する．

スレッドがあるクリティカルセクションを初めて実行を開始し，切り替え判断処理でロックを選択した場合は，そのクリティカルセクションの実行が終了するまでロックを使用する．STM を選択した場合は，ロールバックによって切り替え判断処理に戻るまで STM で動作する．トランザクションがアボートした場合，ロールバック処理の時にスレッド情報リスト内のメンバである「累積アボート数」をアトミックにインクリメントする．

クリティカルセクション終了時，スレッドは関数情報リストの実行中スレッド数をアトミックにデクリメントし，スレッド情報リスト内の実行中の実行中関数ポインタをクリアする．

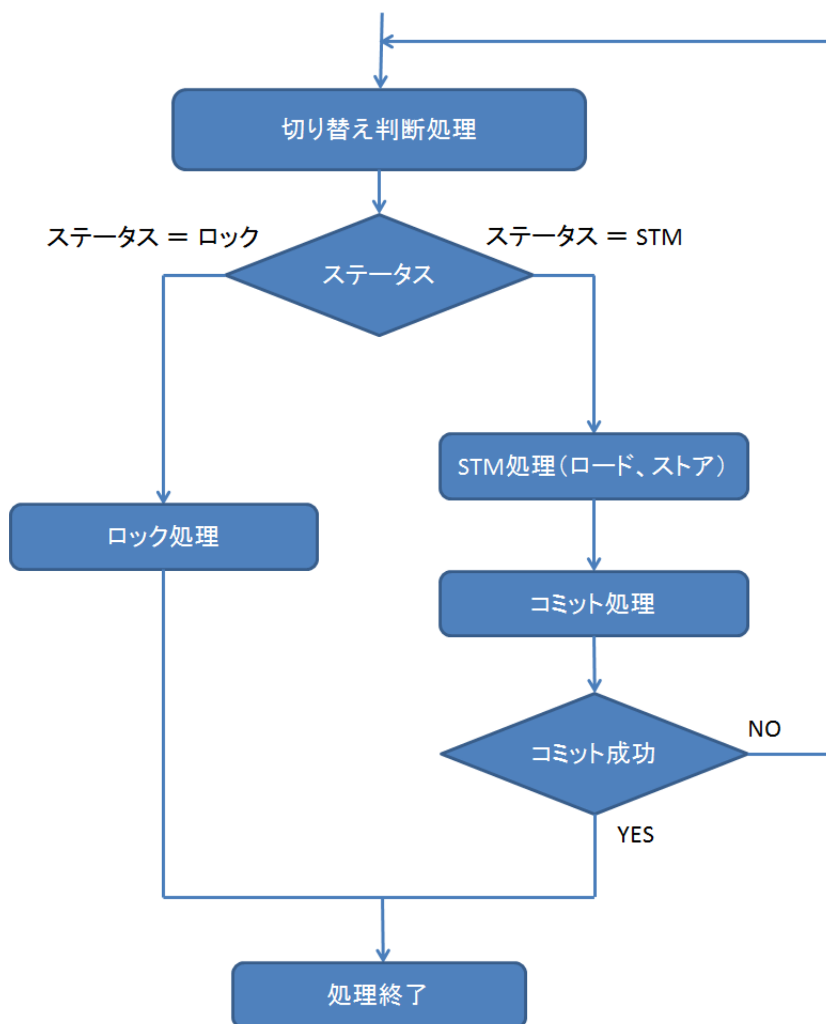


図 4.4: 提案システム内でのスレッドの動作フロー

4.2.6 STM 側の設計

本研究では，STM 側の処理は TinySTM[2] に任せる．本研究はロックと STM との共存と，ロックと STM を切り替えることに重点を置いているため，STM の実装は行わない．

TinySTM では Makefile のパラメータを設定することで TL2 アルゴリズムのセマンティクスを実現することが出来る．本研究では，ロックが共有メモリを獲得していることをトランザクションが認識できる実装でなければならない．したがって，ロックベース STM を使用するのが順当である．

本提案システムでは，ロック側との共存を行う上で矛盾なく排他制御を行うことが出来る STM の実装であるため，TL2 アルゴリズム [5] を採用する．また TL2 の実装は比較的処理量が少ない，様々な状況で安定して排他制御が可能であるため STM 側の動作として適している．

4.2.7 共存の問題

ロックと STM を同じシステム内で動作させることによって問題となるのが共存の問題である．本提案システム内では各スレッドが STM またはロックが有利であると判断した場合に動的に切り替えるため，システム内で STM，ロックが混在することになる．したがって，本提案システムではスレッドが状況に応じてロック，STM を選んで実行するため，システム内で同じ共有メモリに対してロックを選択したスレッドと STM を選択したスレッドが競合を起こす可能性がある．

この状況だと，通常のロックの動作では STM はデータの一貫性を保つことができなくなってしまう．図 4.5 は，ロックと TL2 アルゴリズムで動作する STM が同じ共有メモリに対して変更を加える例である．

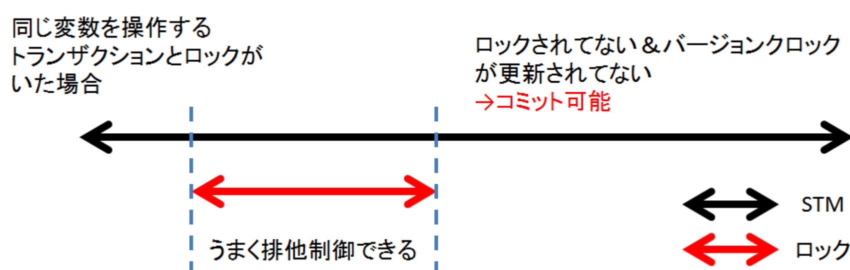


図 4.5: ロックと TL2 STM が競合した場合の例

単純にロックをかけるだけの場合，TL2 アルゴリズムで動作する STM はロックを選択したスレッドがロックを獲得している間に限って，競合を検出することが出来る．しかし，TL2 アルゴリズムは怠惰な競合検出 (lazy conflict detection) であるため，共有メモリのロックを獲得している間に共有メモリに対して読み込み，またはコミットをしない限りは競合を検出することが出来ない．また，ロックを選択したスレッドが排他制御にロックを使用するだけの場合，バージョンクロックの更新を行わないため，STM を選択したスレッドは競合を検出することができない．

4.2.8 ロック側の設計

本研究では，4.2.7 の問題を解決するロックの動作を考察し，設計を行った．以降は特に断りがない限り，本研究で設計・実装したロックを提案手法ロックと記述する．提案手法ロックは，TinySTM のトランザクションとの 4.2.7 節の問題を解決するため，一般的なロックの動作に加えて次の処理を行う．

- グローバルバージョンクロックの更新
- 共有メモリのバージョン管理
- デッドロック回避処理

グローバルバージョンクロックの更新と共有メモリのバージョン管理は，提案手法ロックを使用して排他制御をした後も TinySTM のトランザクションがデータの矛盾を認識できるようにするためである（4.2.7 節）．

デッドロック回避処理は，TinySTM では隣接した 32 ワードの共有メモリ領域を 1 チャンクとして 1 つのロックで管理しているために必要な処理である．

同じチャンク内の共有メモリに対して 2 回ロックを獲得する例を図 4.6 に示す．

スレッドはすでにサイズが 1 ワードの共有変数 A のロックを獲得している．図 4.6 は，A のロックを獲得した状態から同じチャンク内の共有変数 B を取得しようとしているところである．共有変数 B は共有変数 A と同じチャンクであるため，すでにロックを獲得している状態である．しかし，このような状況に対する対策を立てていない場合は，自分自身とのデッドロックが発生してしまう．

TinySTM では，ライトセットに格納された（スレッドローカルで変更された）共有メモリのメタデータの構造体には，対応したロックエントリのアドレスを保存するメンバが存在する．獲得しようとしたロック変数のロックビットが 1 だった場合，ライトセット内のエントリを探索し，すでに自分がロックを獲得していないかチェックすることで自分自身とのデッドロックを回避している．提案手法ロッ

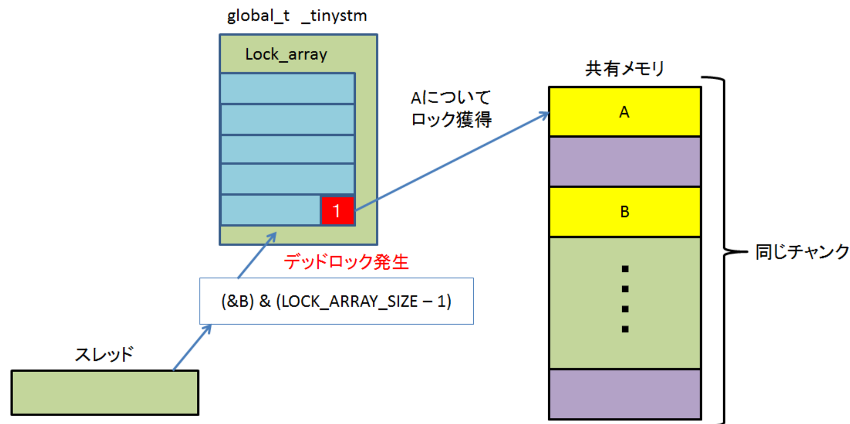


図 4.6: 提案手法ロックのデッドロック例

クでは，このデッドロック回避法を参考に，獲得したロック変数のアドレスを保存するリストデータ構造をスレッドローカルに持たせることで回避する．

なお，このバージョン管理とデッドロック回避処理のオーバヘッドにより，一般的なロック（たとえば *pthread_mutex_lock*）と比較して低速であることに注意する必要がある．特にデッドロック回避処理には，スレッドローカルに保存されたロックの情報リストを探索するため，1 スレッドが扱うロックの数が増えるほどオーバヘッドが大きくなる．したがって，STM と比較した結果が一般的なロックほど性能差が表れないことが考えられる．

4.3 切り替え基準

本システムでは，クリティカルセクション開始時にロックまたは STM のどちらが有利に実行が出来るかを判断する．基本的には STM で実行し，ロールバック発生時にもう一度状況判断を行う．ロックが有利であると判断した場合，当該クリティカルセクション終了までロックを使用して排他制御処理を行う．

具体的には，次のものを基準として判断を行う．

- 当該クリティカルセクションを実行中のスレッド数
- 現在のスレッドのアボート率

- クリティカルセクションが読み込み専用か否か
- クリティカルセクションに入ってからの経過時間

同じクリティカルセクションを実行中のスレッド数を見て，ロールバックする危険性が高いと判断した場合は，ロックを選択する．STM を選択し，アボートして当該クリティカルセクションの開始部分までロールバックした場合，スレッド情報リストの累積アボート数をアトミックにインクリメントし，もう一度切り替え判断処理を行う．クリティカルセクションが読み込み専用の場合，STM は複数のスレッドが同時に実行できるので，ロックよりも速く動作することができる．クリティカルセクションが読み込み専用の場合は，アボート率が一定以下の場合である限りは STM で実行する．また，各クリティカルセクションの実行時間を測定し，平均の実行時間より大幅に遅れる場合にロックに変更するという方法も考えられる．

第 5 章

実装

本提案システムは STM 側の処理に TinySTM を使用している．また，ロックをトランザクションと同じロック変数を使用する必要があるため，TinySTM に本システムを追加する形での実装となる．
具体的には，本研究では次のものを TinySTM に追加した．

- 関数情報リスト
- スレッド情報リスト
- アボート率計算用スレッド
- 切り替え判断器
- ロック処理用の API

以下，これらを順に議論する．

5.1 関数情報リスト・スレッド情報リスト

関数情報リスト，スレッド情報リストの実装は，図 5.1 のような関数情報ノード，スレッド情報ノードの双方向リストで実現する．提案システムでは，このリストがグローバルに 1 つずつ存在する．

各ノードには，それぞれ 4.2.1 節と 4.2.2 節で説明した情報を持っている．関数情報リストのノードは，ユーザが main 関数内で関数を登録する（*register_func* 関数）処理の時に追加される．また，スレッドが生成された時にスレッド情報リストに新しくノードを追加する．
スレッドがクリティカルセクションを始める時，関数情報リストの当該ノードの「実行中のスレッド数」をアトミックにインクリメントする．スレッドが切り

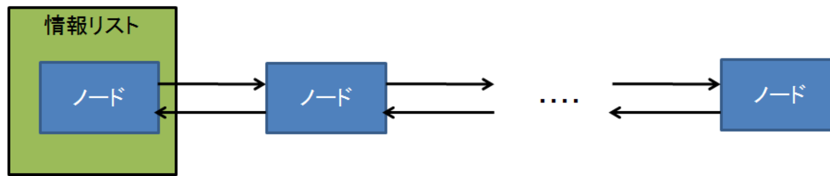


図 5.1: 情報リストのデータ構造

替え判断処理を行う時，関数情報リスト上のこれから行う（またはアボートした）関数の情報ノード，スレッド情報リスト上の自分自身の情報ノードにアクセスし，情報を収集する．

具体的には，これから実行するクリティカルセクションを含んだ関数のポインタをスレッドが保持しておき，クリティカルセクション実行時に関数情報リスト上の同じポインタを持つノードを探索する．クリティカルセクション実行開始時に当該ノード内の実行中のスレッド数をアトミックにインクリメントする．切り替え判断処理の時に，関数情報リスト内の当該ノードの実行中のスレッド数と読み込み専用か否かの情報を収集する．同様に，スレッド情報リスト内のスレッド自身の ID を持ったノードをクリティカルセクション開始時に探索する．ノード内の「実行中の関数ポインタ」部をこれから実行する関数のポインタに書き換える．また，スレッド情報リスト上の当該ノードの開始時間部に現在時刻を書き込む．

切り替え判断処理では，読み込み専用か否かと，単位時間当たりのアボート回数，経過時間を判断材料として切り替えの判断を行う．

5.2 アボート率計算用スレッド

アボート率計算用スレッドは，本システムがユーザープログラムのはじめに初期化された時に生成される．アボート率計算用スレッド内に *structtimeval* 型配列を用意し，スレッド情報リスト内の各ノードに前回アクセスした時間を *gettimeofday* 関数を使用して記録する *structtimeval* 型配列を用意する．配列の個数は，TinySTM で定義されているスレッド数の上限-1（$MAX_THREADS - 1$）だけ用意する．これは，アボート率計算用スレッドの分を差し引いた個数である．このスレッドは常にスレッド情報リストのノードを巡回し，累積アボート数を参照する．

アボート率計算用スレッドは，次の処理を行う．

- アボート率を計算する

- 累積アボート数をクリアする

スレッド情報リスト内の各ノードに対して，累積アボート数と，現在の時間と前回参照した時間との差を利用して単位時間当たりのアボート数を計算する．計算した値をアボート率の場所にアトミックに書き込む．

アボート率計算後，累積アボート数を 0 にアトミックにクリアする．この時，アボートしたスレッドと，累積アボート数についての競合が発生する可能性がある．しかし，アボート率計算用スレッドは 1 つだけであるため，同時に累積アボート数を書き込む可能性は低いと考える．したがって，スレッド情報リストノード内の「累積アボート数」の競合については考慮しない．

5.3 切り替え判断器

スレッドはクリティカルセクションのはじめに切り替え判断器を使用して STM，ロックのどちらで実行するのが有利かを判断する．切り替え判断器の実体は，クリティカルセクションが読み込み専用か否かとアボート率，経過時間などの情報を各情報リストから収集し，判断結果を戻り値として返す関数である．

アボート率，経過時間の閾値を用意し，閾値以内であれば STM，閾値を超えている場合はロックと判断する．切り替え判断器はソースコード 5.1 のようなコードで実現される．

ソースコード 5.1: 切り替え判断器のコード

```
#define STATUS_STM 0
#define STATUS_LOCK 1

int switch_mutex( void *func, int id )
{
 //関数が読み込み専用か否かを保存する変数
 int func_ro;
 //同じ関数を実行中のスレッド数
 int num_threads;
 //アボート率, 経過時間を保存する変数
 double rate_abort;
 struct timeval *exec_time;
 //関数情報リストから読み込み専用か否かを取得
 func_ro = get_ro( func );
 //関数情報リストから同じ関数を実行している数を取得
 num_threads = get_num_threads( func );
 //スレッド情報リストから情報を取得
 rate_abort = get_rate_abort( id );
 //経過時間を取得
 exec_time = get_exec_time( id );

 //切り替えの判断
 if( func_ro ){
  if( rate_abort < アボート率閾値RO
    && exec_time < 経過時間閾値
    && num_threads < スレッド数閾値 ){
     return STATUS_STM;
  }else{
     return STATUS_LOCK;
  }
 }else{
  if( rate_abort < アボート率閾値RW
    && exec_time < 経過時間閾値
    && num_threads < スレッド数閾値 ){
     return STATUS_STM;
  }else{
     return STATUS_LOCK;
  }
 }
}
```

各情報リストから，読み込み専用か否か，同じ関数を実行中のスレッドの数，アボート率，開始時間を取得し，経過時間は現在の時間との差を *get_exec_time* 関数内で計算したものを取得する．読み込み専用か否かによって，アボート率閾値が若干異なる．ソースコード 5.1 では，アボート率閾値 RO，アボート率閾値 RW と表記している．これは，読み込み専用のトランザクションは処理量が少ないため，アボート率が読み込み専用でないトランザクションのアボート率よりも一定以上高くても高速に動作することが期待できる．

切り替え判断器で取得したステータスを元に，ロック，STM のコードを実行する．具体的には，ソースコード 5.2 のようになる．

ソースコード 5.2: 本システムでの排他制御の実行開始例

```
jmp_buf _e;
//本システムで排他制御の開始時に実行する関数
void hymutex_start(){

 //これから実行するクリティカルセクションを含んだ
 //関数ポインタを取得
 void *func = get_func();

 //ロールバック時は_eを使用してここまでジャンプする
 sigsetjmp( _e );

 if( switch_mutex( func, pthread_self() ) == STATUS_LOCK ){
  /*ロックのコードを実行開始*/
 }else{
  /*STMのコードを実行開始*/
 }
}
```

TinySTM では，ロールバックに sigsetjmp，siglongjmp を使用している．ロールバックを行うとき，スレッド情報リストの累積アボート回数をインクリメントしてソースコード 5.2 内 10 行目にロールバックする．切り替え処理判断処理を実行し，戻り値に応じてロックコード，STM コードの実行を開始する．

5.4 ロック処理用の API

本研究では，提案手法ロックの動作をまとめたライブラリを作成した．これは，本提案システムでスレッドがロックを選択した場合，使用するロックの API である．

提案手法ロックは，次のようにロックを実行する．

1. 対応したロック変数のエントリの算出
2. ロックビットの確認
3. ロックの獲得
4. ロック獲得リストへのロックエントリの登録

スレッドはまず，ソースコード 4.1 の計算式を使用して当該共有メモリに対応したロックエントリを計算する．ロックエントリのロックビットを確認し，ロックビットが 1 だった場合は自分のロック獲得リスト内のノードに当該ロックエントリのアドレスが登録されていないかを確認する．ロック獲得リストは，スレッドがローカルに持っている図 5.1 と同じ双方向リスト構造体で実現されたデータ構造である．ノードには獲得したロックエントリのアドレスを格納する．ロック獲得リスト内にロックエントリのアドレスがなければスピンし，リスト内にロックエン

トリのアドレスが登録されていたらロック処理を終了する．ロックビットが 0 であれば，コンペアアンドスワップを使用して，ロックビットを確認した時と同じ値であれば最後尾ビットを 1 にした値をロックエントリにアトミックに書き込む．エントリへのアトミックな書き込みが完了した後，ロックエントリのアドレスをロック獲得リストに登録する．

アンロックは，次の手順で行う．

1. ロック獲得リストからロックエントリを削除
2. バージョンクロックの更新
3. 新しいバージョンをロックエントリへ書き込む

アンロックするロックエントリをロック獲得リストから削除し，グローバルバージョンクロックをインクリメントアンドフェッチを使用して更新する．フェッチしてきた値に 1 を加算（グローバルバージョンクロックと同じ値にするため）し，1 ビット左論理シフトしたものをロックエントリに書き込むことでアンロックする．

第 6 章

評価と考察

6.1　実験環境

評価で使用した環境は次のとおりである.

- CPU: Intel Xeon 2.8GHz 2 基 (4 コア + 4 コア)
- OS: Mac OS X 10.8.0
- メモリ:12GB

6.2　予備実験

本研究では，次の実験を予備実験として行った.

- 提案手法ロックの性能調査
- 切り替え基準の検証

6.3　提案手法ロックの評価

本研究では *pthread_mutex_lock* と比較するため 5.4 節で示した TinySTM 上でトランザクションと共存させるロックライブラリを作成した（5.4 節）. 提案手法ロックは 4.2.8 節で示したとおり，デッドロック回避処理と共有メモリのバージョン管理を行うため，*pthread_mutex_lock* などの一般的なロックと比較して低速である.

実験の内容は，1 つのスレッドがロック，アンロックを 1000 回から 10000 回繰り返すというものである. 妥当性を示すため，獲得したロック変数のリストの操作を除いたもの（提案ロック（シンプル)）とも比較を行う. この実験の結果を図 6.1 に示す.

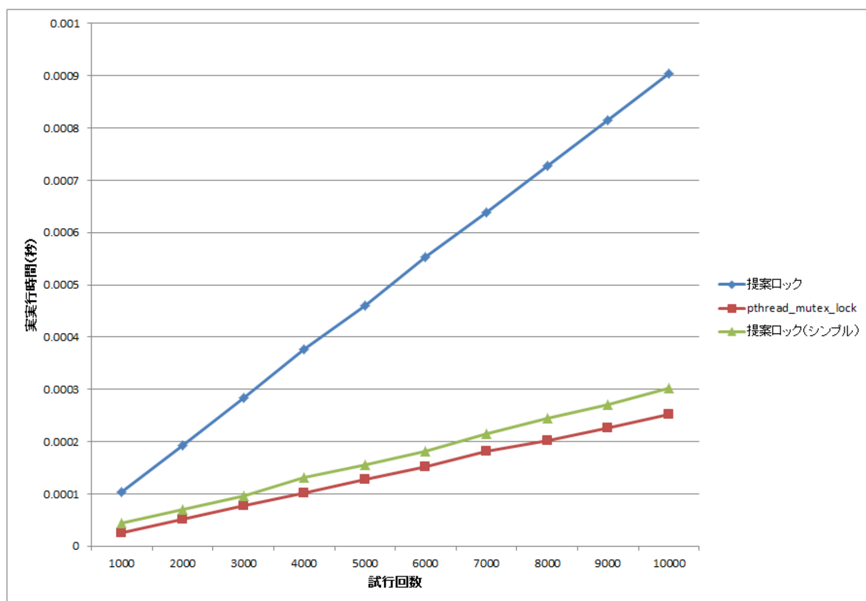


図 6.1: *pthread_mutex_lock* と提案手法ロックの性能比較

このグラフを見ると，提案手法ロックは *pthread_mutex_lock* と比較しておよそ 4 倍時間がかかっていることが分かる．これは予想のとおり，STM の動作（ロック変数のバージョンフィールドの更新，獲得したロックリストの操作）にオーバヘッドがかかっているためである．根拠として，獲得したロック操作を行わない提案手法ロック（提案手法ロック（シンプル）と表記）は *pthread_mutex_lock* と比較してわずかに実行時間が増加しているにとどまっている．したがって，本提案手法ロックの性能は図 6.1 のとおりの性能で妥当である．

5.4 節で説明したとおり，これは TinySTM の実装にしたがっているため，デッドロック回避処理を含んでいる．ロック変数の粒度を共有メモリ数と同じにすることによって，このデッドロック回避処理が必要なくなるため，高速化が望まれるが，その代わりに共有メモリ数だけロック変数が必要になるので，共有メモリが多いシステムには適さない．

6.4 切り替え基準の決定

本研究では，ロック，STM を使用したアプリケーションの性能は表 4.1 で示した状況の時，高く（低く）なるという前提である．それを実証するため，予備実験を行った．予備実験では，同じ操作を行うアプリケーションの提案手法ロック版，STM 版を作成し，アプリケーションのスループットを検証した．

前提の検証

予備実験では，次の 3 種類の実験を行った．

1. 共有メモリ 1 箇所に対しての競合頻度とロック（STM）の性能との関係を調べる実験
2. クリティカルセクション内で触る共有メモリの数とロック（STM）の性能との関係を調べる実験
3. クリティカルセクション内でスレッドローカルに行う計算処理量とロック（STM）の性能との関係を調べる実験

同じ共有メモリに対して，書き込み競合が多い場合は STM ではアボートが頻発するため，ロックが有利である．逆に，同じ共有メモリに対して読み込み競合が多い場合は STM は並列に実行できるため，STM が有利である．この前提が正しいことを実証するため，共有メモリ 1 箇所に対して書き込み競合が多い状況と読

み込み競合が多い状況を作り出して，それぞれでロック，STM が高速になることを実証する．

クリティカルセクション内で多くの共有メモリを使用する場合，使用した数だけ他トランザクションとの競合のリスクが高まる．したがって，多くの共有メモリをクリティカルセクション内で使用する場合，ロックが有利である．クリティカルセクション内で使用する共有メモリ数が少ない場合は，アボートのリスクが少なくなるため，STM が有利だと考えられる．しかし，前述の競合状況にも依存するため，一概に STM が有利になるとは限らない．

クリティカルセクションが長い場合，たとえば共有メモリを使用して微積分を行い，書き戻す場合などでは，STM はクリティカルセクションを実行する時間が長くなるほど競合を起こすリスクが高まる．したがって，ロックが有利である可能性が高い．反対に，クリティカルセクションが短い場合はアボートのリスクが少なく，STM が有利であると考えられるが，これも前述の競合状態に依存するため，一概に言うことは出来ない．

以上の前提を実証するため，それぞれの実験で各共有メモリのすべての操作回数のうち，90% が書き込み操作のものと，90% が読み込みのベンチマークを作成した．

競合頻度と性能の関係の検証

共有メモリ 1 つに対しての書き込み競合，読み込み競合の頻度とロック，STM の性能の関係を検証し，切り替え基準を検討する．

実験概要　この実験では，共有メモリを 1 つだけ用意し，共有メモリに対して 1 つから 8 つのスレッドが読み込み，または書き込みを行う．コードはロック，STM それぞれソースコード 6.1 とソースコード 6.2 のようになる．

ソースコード 6.1: 共有メモリにアクセスするロックのソース

```
#define NUM_THREADS      1~8;
#define RATE_WRITE       10 or 90;
#define DENOMINATOR      100;
#define NUM_THREAD_LOOP   10000;

//共有メモリ
stm_word_t shared_data;


//排他制御を含んだ関数
void *critical_section( void )
{
 int i;
 stm_word_t tmp;
 for( i = 0; i < NUM_THREAD_LOOP; i++ ){
  //任意時間停止
  SLEEP_THREAD;
  if( rand() % DENOMINATOR < RATE_WRITE ){
   //書き込みを行う場合のクリティカルセクション
   lock_at( &shared_data );
   SLEEP_CS;
   //本実験ではインクリメントを行うだけにする
   tmp = shared_data;
   tmp++;
   shared_data = tmp;
   unlock_at( &shared_data );
  }else{
   //読み込みだけ行う場合のクリティカルセクション
   lock_at( &shared_data );
   SLEEP_CS;
   //読み込みだけ行う
   tmp = shared_data;
   unlock_at( &shared_data );
  }
 }
}
```

ソースコード 6.2: 共有メモリにアクセスする STM のソース

```
#define NUM_THREADS      1~8;
#define RATE_WRITE       10 or 90;
#define DENOMINATOR      100;
#define NUM_THREAD_LOOP   10000;

//共有メモリ
stm_word_t shared_data;


//排他制御を含んだ関数
void *critical_section( void )
{
 int i;
 stm_word_t tmp;
 for( i = 0; i < NUM_THREAD_LOOP; i++ ){
  //任意時間停止
  SLEEP_THREAD;
  if( rand() % DENOMINATOR < RATE_WRITE ){
   //書き込みを行う場合のクリティカルセクション
   TM_START;
   SLEEP_CS;
   //本実験ではインクリメントを行うだけにする
   tmp = TM_LOAD( &shared_data );
   tmp++;
   TM_STORE( &shared_data, tmp );
   TM_COMMIT;
  }else{
   //読み込みだけ行う場合のクリティカルセクション
   TM_START;
   SLEEP_CS;
   //読み込みだけ行う
   tmp = TM_LOAD( &shared_data );
   TM_COMMIT
  }
 }
}
```

1 つだけある共有メモリに対して，書き込みを行う場合はスレッドローカル変数 tmp に値を読み込み，tmp をインクリメントし，tmp の値を共有メモリに書き込んでいる．読み込みを行うだけの場合は，tmp に共有メモリの値を読み込み，クリティカルセクションを終了する．このようなクリティカルセクションをスレッドは *NUM_THREAD_LOOP* 個だけ持っており，今回はすべての実験を通してこのマクロを 10000 に固定する．つまり，スレッドは 10000 個のクリティカルセクションを実行することになる．スレッドは 1 割，または 9 割の確率で共有メモリに対して書き込みを行う．

マクロ *SLEEP_THREAD* と *SLEEP_CS* は，任意に設定した休止時間 *SLEEP_TIME_THREAD* と *SLEEP_TIME_CS* の数だけ for ループを実行して，実行時間を増やすものである．これはソースコードのように定義されている．

ソースコード 6.3: 休止マクロの定義

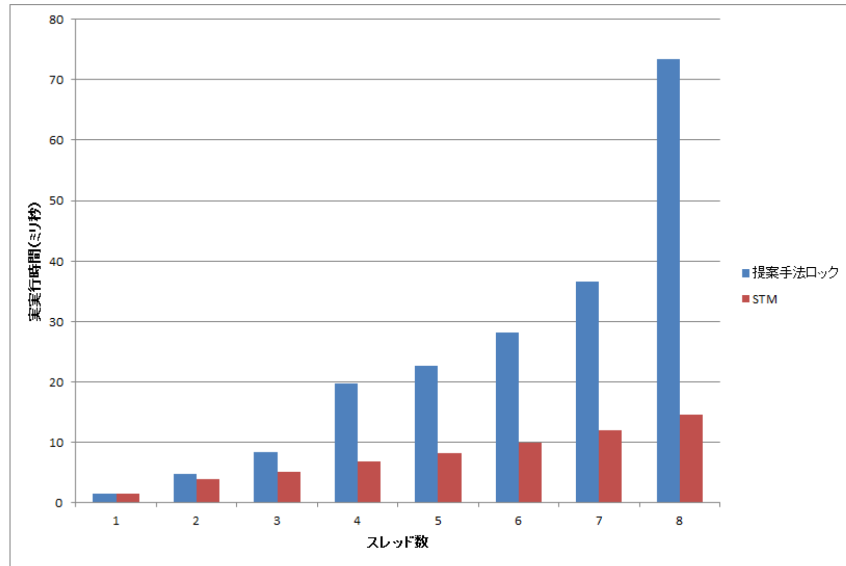
```
1 #define SLEEP_THREAD volatile long sleep_thread; for( sleep_thread = 0; sleep_thread <
      SLEEP_IME_THREAD; sleep_thread++ ){}
2 #define SLEEP_CS volatile long sleep_cs; for( sleep_cs = 0; sleep_cs < SLEEP_TIME_CS;
      sleep_cs++ ){}
```

これを利用して，スレッド全体の実行時間とクリティカルセクションの実行時間のバランスをとる．本節の実験では，各ループ時間を調節して，書き込みが多い状況では 1 スレッドが 1.7 ミリ秒前後で，読み込みが多い状況ではスレッドが 1.5 ミリ秒で実行するように調整して実験した．

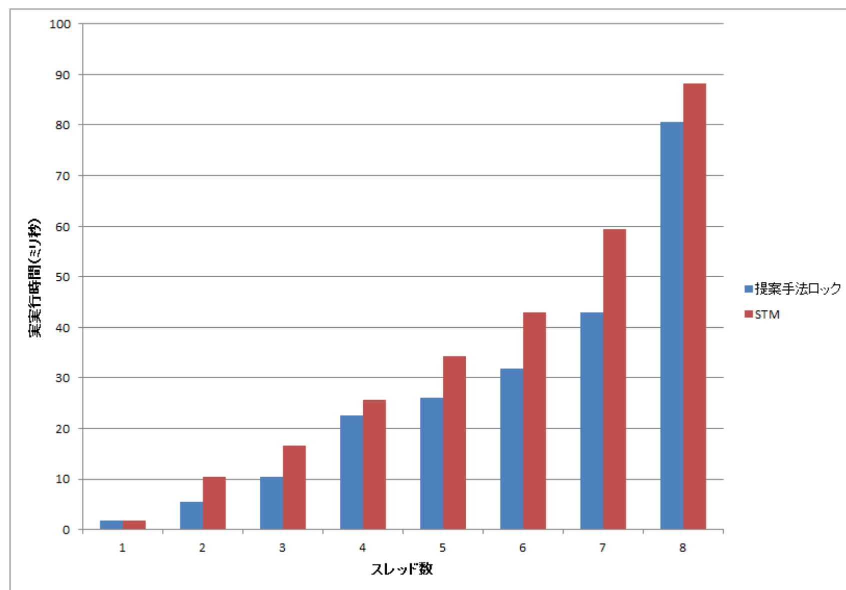
書き込みするクリティカルセクションは読み込みを行うクリティカルセクションより実行時間が遅いため，書き込みが 200 マイクロ秒程度遅く調節されている．スレッド実行時間全体に占めるクリティカルセクション実行時間の割合は，読み込みが多い状況，書き込みが多い状況ともに 6 割程度となっている．一般的なアプリケーションで，スレッド実行時間中に占めるクリティカルセクション実行時間の割合が 6 割程度あるものは考えづらいが，本節の実験では競合を意図的に頻発させるためであるので，クリティカルセクション前のループ（*SLEEP_THREAD*) による休止は行わないようにしている．クリティカルセクション前のループが一定以上行われると，スレッド間にクリティカルセクション実行開始のタイミングにばらつきが大きくなり，競合が期待より発生しないためである．

実験結果　この実験を行った結果を図 6.2 に示す．

読み込みが 9 割の時，STM はパラレルに実行することができ，スレッド数が増えるにつれてロックよりスループットが高いことがわかる．これは 2.2.1 で説明した状況が頻発し，STM は 8 スレッドの時 80000 回のクリティカルセクションのうち 30000 回程度アボートしているものの，パラレルに実行出来ている割合が多いのでロックよりスループットが良いことが実証された．



(a) 読み込み割合が 90%の場合



(b) 書き込み割合が 90%の場合

図 6.2: 高競合時の提案手法ロックと STM の性能比較

書き込みが多い状況では，2 スレッドの時にスループットが提案手法ロックの 2 分の 1 程度であったが，8 スレッドの時にスループットの差が 10 %程度となっている．STM のアボート回数も，4 スレッドの時に 34000 回程度，8 スレッドの時に 140000 回程度となっているが，提案手法ロックについては，デッドロック回避処理などのオーバヘッドが影響し，スループットが低下したと考えられる．予想のとおり，書き込み競合が多い時はロックの方がスループットが良かったが，予想していたものよりも STM のスループットとの差は開かなかった．

以上のことから，競合頻度については読み込み競合が多い状況の場合は STM，書き込み競合が多い場合はロックを使用するべきであるという前提は正しいと考える．

6.4.1　共有メモリ数と性能の関係の検証

実験概要　本節では，共有メモリを 10 から 100 個の配列に変更し，スレッドがすべての共有メモリに対して操作するプログラムを作成した．

ソースコード 6.4: 共有配列にアクセスするロックのソース

```
//共有メモリ
stm_word_t shared_data[ NUM_SHARED_DATA ];
//排他制御を含んだ関数
void *critical_section( void )
{
 int i;
 stm_word_t tmp;
 for( i = 0; i < NUM_THREAD_LOOP; i++ ){
  //任意時間停止
  SLEEP_THREAD;
  for( lock = 0; lock < NUM_SHARED_DATAS;
      lock++ ){
   lock_at( list, &shared_data[ lock ] );
  }
  SLEEP_CS;
 if( rand() % DENOMINATOR < RATE_WRITE ){
  for( i = 0; i < NUM_SHARED_DATAS; i++ ){
   tmp = shared_data[ i ];
   tmp++;
   shared_data[ i ] = tmp;
  }
 }else{
  for( i = 0; i < NUM_SHARED_DATAS; i++ ){
   tmp = shared_data[ i ];
  }
 }
 for( lock = NUM_SHARED_DATAS - 1; lock >=
     0; lock-- ){
  unlock_at( list, &shared_data[ lock ] );
 }
}
```

ソースコード 6.5: 共有配列にアクセスする STM のソース

```
//共有メモリ
stm_word_t shared_data[ NUM_SHARED_DATAS ];
//排他制御を含んだ関数
void *critical_section( void )
{
 int i,j;
 stm_word_t tmp;
 for( i = 0; i < NUM_THREAD_LOOP; i++ ){
  if( rand() % DENOMINATOR < RATE_WRITE ){
   TM_START( RW );
   SLEEP_CS;
   for( j = 0; j < NUM_SHARED_DATAS; j++ ){
    tmp = TM_LOAD( &shared_data[ j ] );
    tmp++;
    TM_STORE( &shared_data[ j ], tmp );
  }
  TM_COMMIT
 }else{
  TM_START( RO );
  SLEEP_CS;
  for( j = 0; j < NUM_SHARED_DATAS; j++ ){
   tmp = TM_LOAD( &shared_data[ j ] );
  }
  TM_COMMIT
 }
}
```

このプログラムでは，スレッドは$RATE_WRITE$%の確率ですべての共有メモリの値をインクリメントするか，$100 - RATE_WRITE$% の確率ですべての共有メモリの値を読み込む．6.4 節と異なる点は，次のとおりである．

- 共有メモリが配列になっている
- スレッド数が 8 に固定されている
- スレッド実行時間，クリティカルセクション実行時間のバランス

共有メモリを配列にし，10 から 100 の範囲で共有メモリの個数を変動させて性能がどのように示されるのかを検証する．また，この実験ではスレッド数を 8 に固定し，ロック，STM ともにスレッド実行時間のうちクリティカルセクションの時間を 1%に固定している．

予想では，表 4.1 のとおり，クリティカルセクション内で使用する共有メモリの数が少なければ STM の方が有利であり，共有メモリの数が多い場合はアボートが頻発するようになるため，ロックの方が性能が高いというものである．

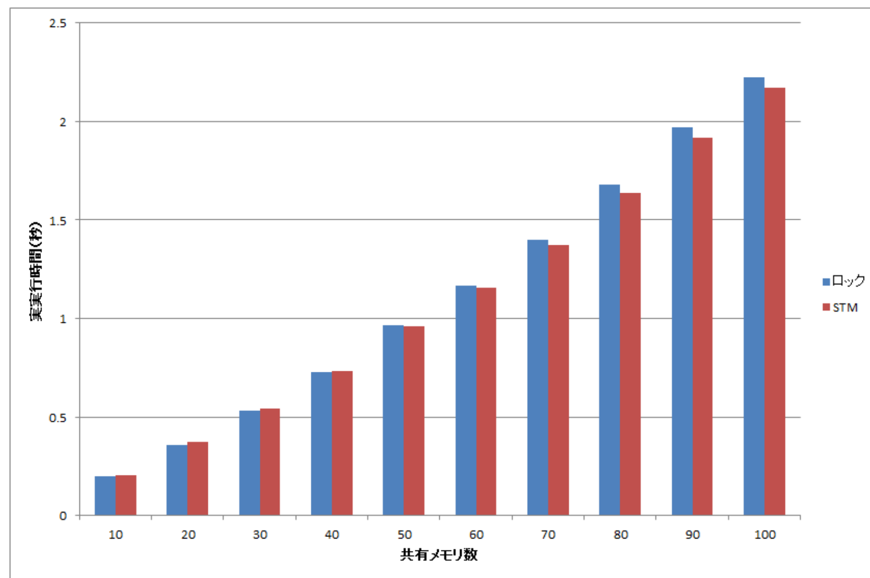
実験結果　実験の結果，図 6.3 のような実実行時間の推移があった．

おおむね予想のとおり，読み込みが多い場合は STM のスループットが高く，書き込みが多い場合はロックのスループットが高いことが分かった．しかし，読み込みが多い状況でも STM はロールバックが頻発し，ロックよりスループットが高いものの，その差は最も大きいところでも 5%程度である．一方，書き込みが多い場合では，STM はアボートが頻発しており，ロックは常に STM より 2 倍以上スループットが高い．

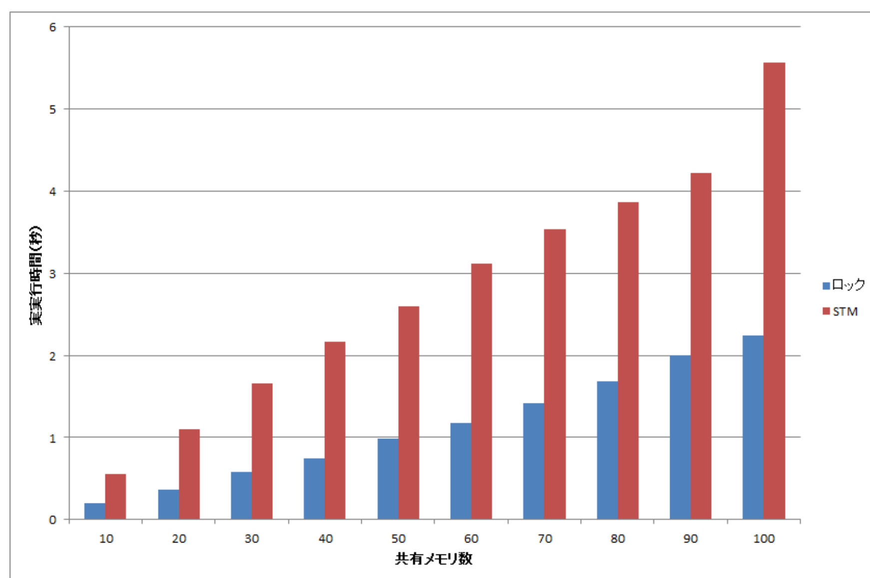
6.4.2　クリティカルセクションの割合と性能の関係の検証

実験概要　これは，6.4 節のプログラム内のループ回数マクロ$SLEEP_TIME_THREAD$，および$SLEEP_TIME_CS$ の値を変動させることで，スレッド実行時間中のクリティカルセクション実行時間を 1%，5%，10%に変動させる．スレッド実行時間を 190 ミリ秒に固定し，クリティカルセクション実行時間を 1.9 ミリ秒（1%），3.8 ミリ秒（5%），19 ミリ秒（10%）になるよう調節し，測定を行った．

予想では，クリティカルセクションの時間が短い場合は，スレッド数が増加しても STM のアボート率が低いため，ロックよりスループットが高い．クリティカルセクションの時間が長くなると，ロールバックの頻度が高くなるためにロックのスループットが高くなると考えている．



(a) 読み込み割合が 90%の場合



(b) 書き込み割合が 90%の場合

図 6.3: クリティカルセクション内で使用する共有メモリ数と性能の関係

実験結果　実験を行った結果，図 6.4，6.5，6.6 のようになった．

予想のとおり，クリティカルセクションの割合が大きくなるにつれて，読み込みが多い場合は STM のスループットが高く，書き込みが多い場合ではロックのスループットが高いことがグラフからわかる．また，クリティカルセクションの割合が大きくなるほど，ロックの競合率も高くなり，増加傾向にある．この増加率は STM の増加率よりも大きく，クリティカルセクションの割合が 20%になると，書き込みが多い状況でロックのスループットが STM より低下した．これは 1 割の読み込みトランザクションが STM ではパラレルに実行できた分だけロックより高速に実行できたと考えている．

6.5　評価実験

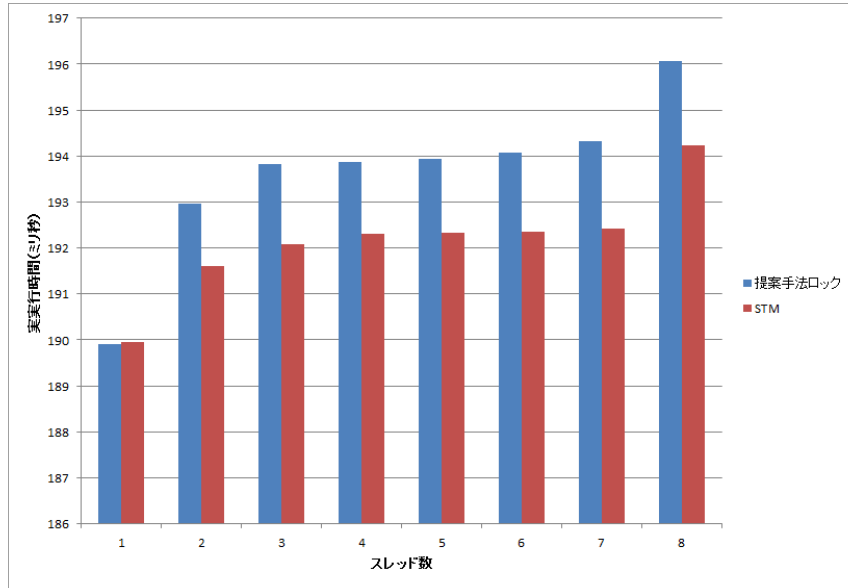
6.4 節の実験結果を元に，ロックが有利と考えられる処理を実行する時に提案手法ロック，STM が有利だと考えられる処理を実行する時に STM に切り替わるプログラムを作成した．このプログラムでは手動で切り替えを行う．つまり，切り替えはスレッドが状況を判断して行うのではない．したがって，切り替えのオーバヘッドは 0 である．読み込みをする時はトランザクション，書き込みをする時は提案手法ロックに切り替えるようコードを記述した．

本提案システムのプロトタイプとロック，STM とスループットの比較を行った．評価プログラムは，6.4 節で行ったものと同じである．これは，他ベンチマークプログラムでは本提案システムのスレッドはすべてロック，または STM で固定されてしまうからである．つまり，1 つだけの共有メモリに対して 1 万個のクリティカルセクションを持つスレッドが操作を行うものである．スレッド全体の実行時間に対してクリティカルセクションの実行時間を 1%になるよう調節し，読み込みが多い場合は 177 ミリ秒，書き込みが多い場合は 192 ミリ秒程度になるよう調節して性能比較を行った．

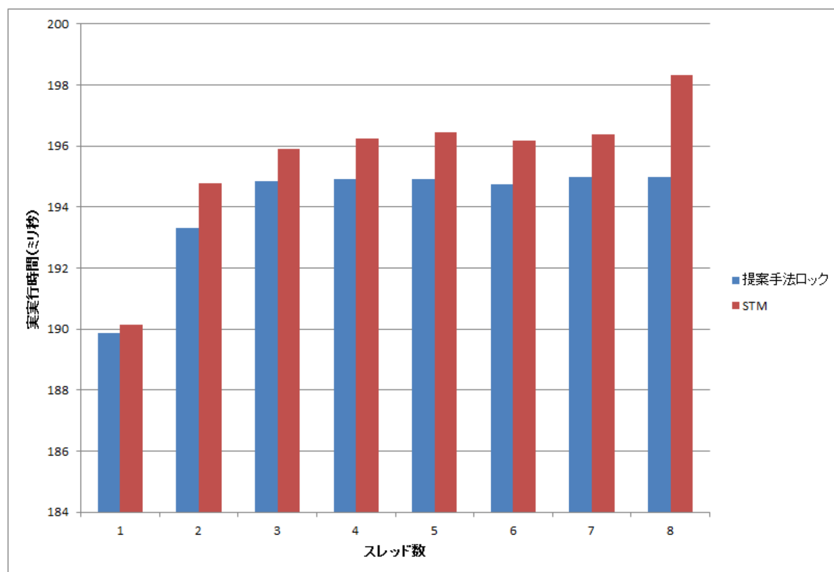
実験の結果，図 6.7 のようなグラフの結果となった．

評価の結果，本提案システムは読み込みが多い場合は提案手法ロック単体よりスループットが高く，書き込みが多い場合は提案手法ロック単体，STM 単体よりもスループットが高いことが分かった．

これは読み込みが多い状況の場合，読み込み処理をトランザクションで実行しているため，コア数だけ同時に実行することが出来ていることが要因となり，提案手法ロックより高速にアプリケーションを実行出来たのだと考えられる．しかし，スレッドが書き込む時はクリティカルセクションのはじめにロックを獲得してしまい，他スレッドの読み込みトランザクションがアボートしてしまうため，STM

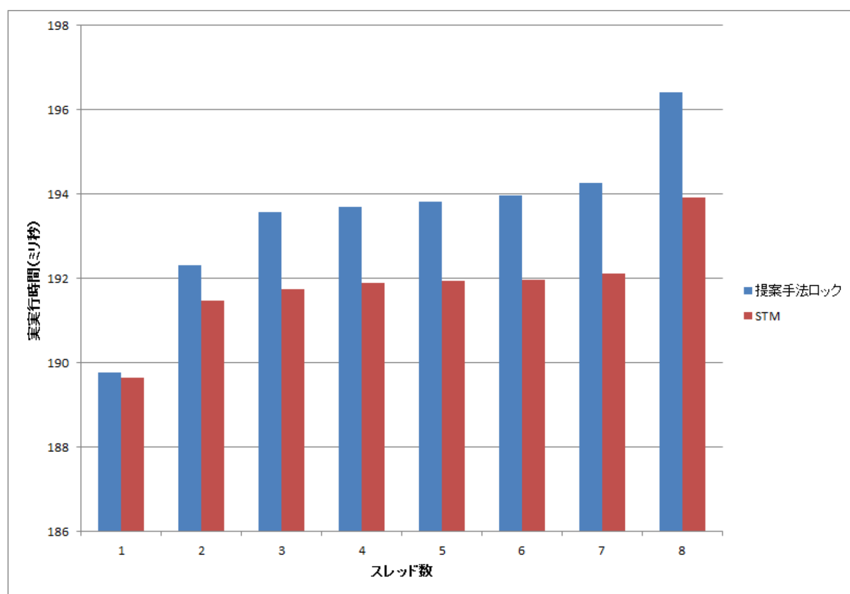


(a) 読み込み割合が 90%の場合

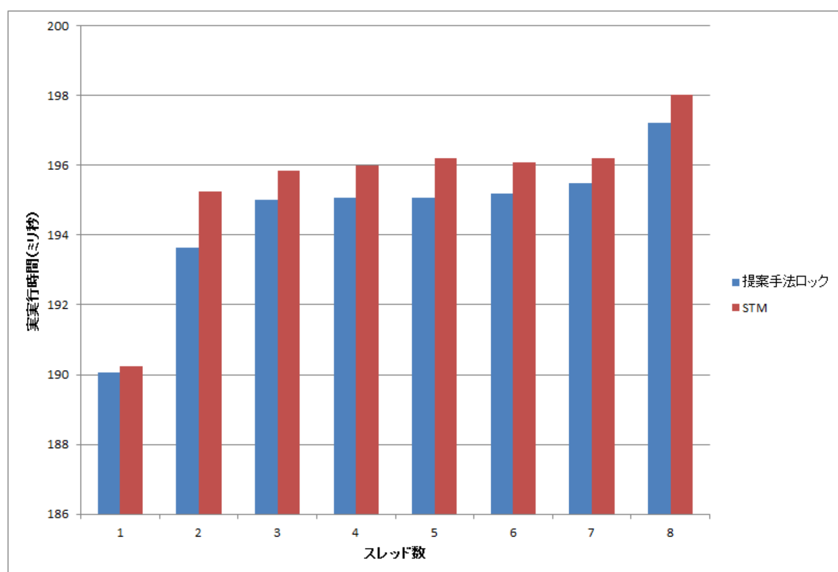


(b) 書き込み割合が 90%の場合

図 6.4: クリティカルセクションの実行割合が 1%の時の性能比較

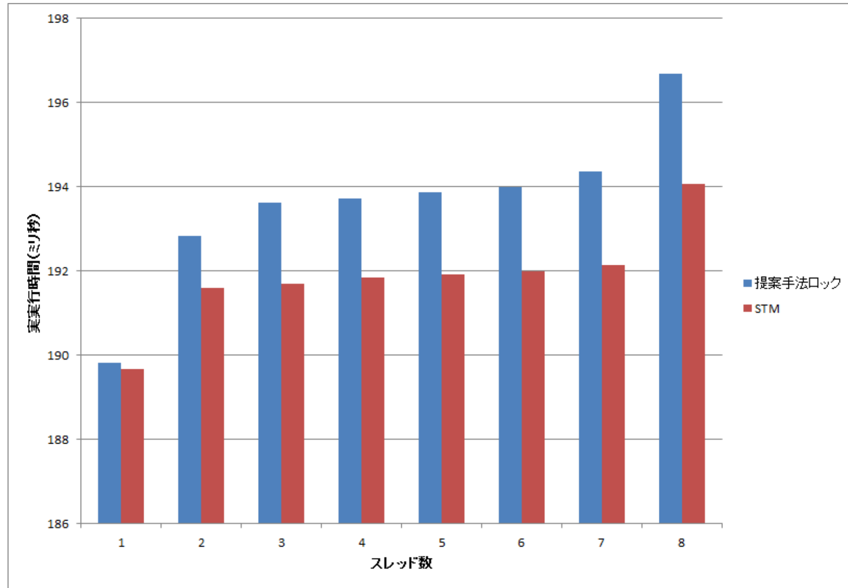


(a) 読み込み割合が 90%の場合

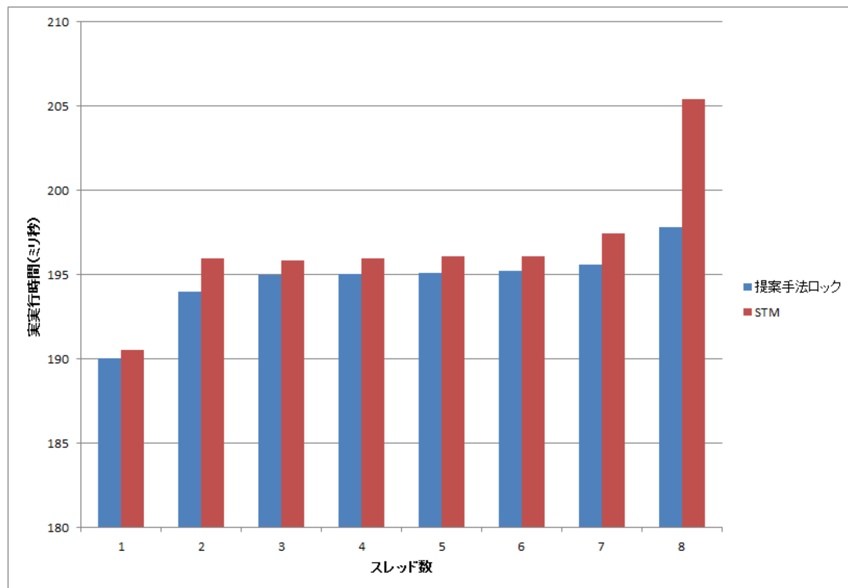


(b) 書き込み割合が 90%の場合

図 6.5: クリティカルセクションの実行割合が 5%の時の性能比較

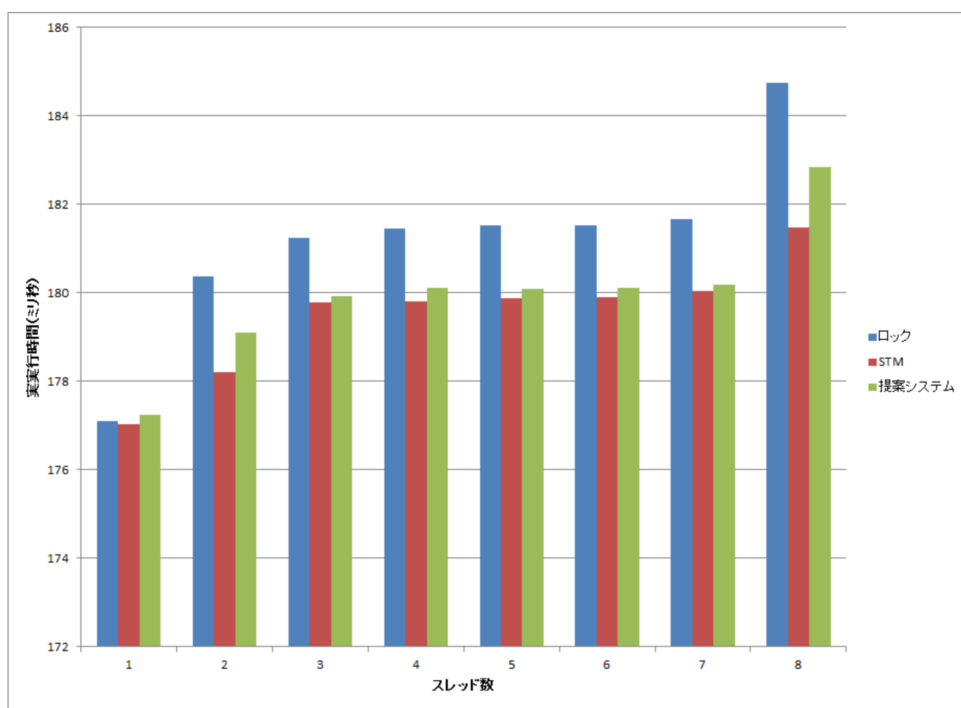


(a) 読み込み割合が 90%の場合

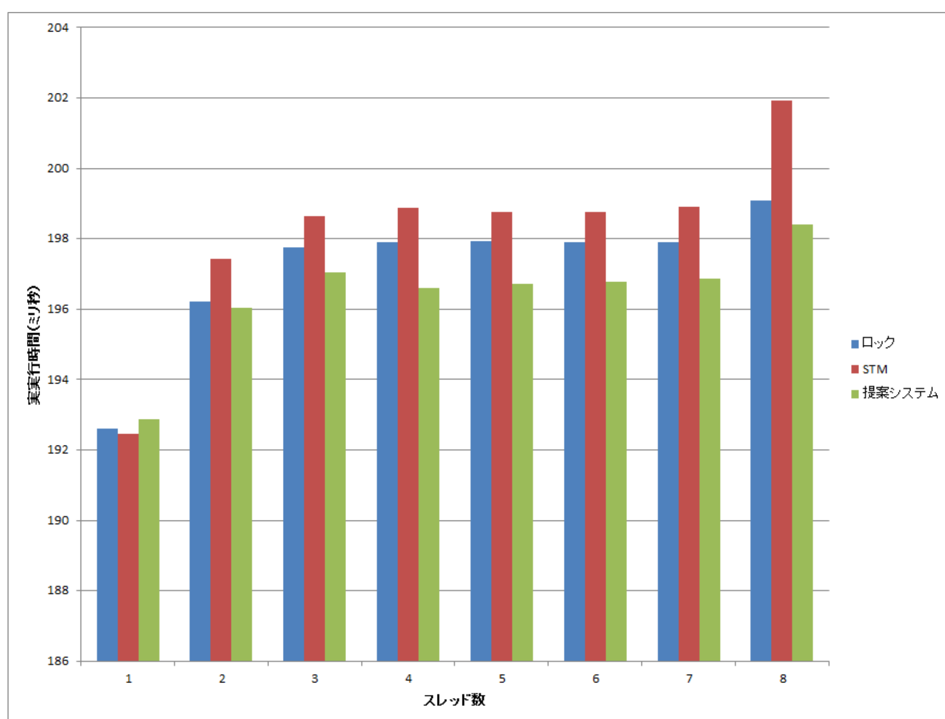


(b) 書き込み割合が 90%の場合

図 6.6: クリティカルセクションの実行割合が 10%の時の性能比較



(a) 読み込み割合が 90%の場合



(b) 書き込み割合が 90%の場合

図 6.7: 評価実験

よりスループットが劣る．

書き込みが多い場合は，アボートするリスクが高い書き込みトランザクションを使用せず，ロックを使用したことでSTMよりスループットが高くなったと考えられる．また，読み込みは軽量なトランザクションを使用することで，ロックよりも高速な処理を実現したと思われる．

今回は実装が出来なかったが，実際にSTMライクにコーディングが出来るようになった上で上記の性能を発揮することが出来れば，本システムはロック単体，STM単体より優位であると考える．

第 7 章

結論

本研究では，ロックと STM を動的に切り替える排他制御システムを提案した．ユーザは STM ライクにコードを記述することができ，デッドロックや合成性を意識することなくコードを記述できるという点と，実行中の関数やスレッドの状況を見て動的にロック（STM）に切り替えることでパフォーマンスの向上が期待できることを示した．

また，システムの設計，実装を示し，プロトタイプとロック単体，STM 単体との性能比較を行って，本システムのパフォーマンス上の優位性を示した．

今後の課題としては，次の 3 つが挙げられる．

- 提案手法ロックの性能改善
- コンパイラの拡張
- 切り替えの自動化

提案手法ロックは TinySTM のロック粒度が 32 エントリ毎であったため，デッドロック回避処理が必要であった．このオーバヘッドによる影響で，提案手法ロックは一般的なロックよりも性能が劣っている．ロック粒度を改善するには，TinySTM 内のロック変数の数を増やす必要があるが，あまり数を多くするとロックベース STM の利点であるキャッシュローカリティがなくなり，多くの共有メモリを使用するマルチスレッドアプリケーション時には性能の低下が予想される．

現段階で本研究で作成したシステムでは，ユーザが STM ライクに記述したコードからロックコードを生成することはできない．コンパイラの機能を拡張することで提案手法ロックを使用するロックコードを生成する．

本研究ではロック，STM の切り替え処理は手動で行った．つまり，スレッドは状況を動的に判断はしていない．今後はこの処理をスレッドが行うよう，5 章で述べた実装を進める．

参考文献

[1] Nir Shavit, and Dan Touitou, "Software Transactional Memory," ACM PODC '95 Symposium on the Principles Of Distributed Computing, pp. 204–213, 1995.

[2] TinySTM TMware.org
URL: http://www.tmware.org/tinystm

[3] Calin Cascaval, Colin Blundell, Maged Michael et al. "Software Transactional Memory: Why Is It a only Research Toy?" Queue Volume 6 Issue 5, pp.46–58, ACM, 2008.

[4] Tim Harris and James R. Larus, and Ravi Rajawar, "Transactional Memory 2nd Edition," Morgan & Claypool Puglishers, ISBN:9781608452354, 2010.

[5] Dave Dice, Ori Shalev, and Nir Shavit, "Transactional Locking II, " ACM DISC ' 06: Proceedings of the 20th international conference on Distributed Computing, pp.194–208, 2006.

[6] Mourice Herlihy, and Nir Shavit,(株) クイープ訳 "the Art of Multiprocessor Programming 並行プログラミングの原理から実践まで," アスキー・メディアワークス, ISBN:9784048679886, 2009.

[7] Eric Koskinen, and Maurice Herlihy, "Dreadlocks: Efficient Deadlock Detection," ACM SPAA'08: Proceedings of the 20th annual Symposium on Parallelism in Algorithms and Architectures, pp.297–303, 2008.

[8] Robert Ennals, "Software Transactional Memory Should Not Be Obstruction Free," Intel Research at Cambridge, 2005.

[9] Maurice Herlihy and Sun Ye, "Distributed Transactional Memory for Metric-space Networks," ACM DISC'05 Proceedings of the 19th International Conference on Distributed Computing, 2005.

[10] Jennifer Mankin, David Kaeli, and John Ardini, "Software Transactional Memory for Multicore Embedded System," ACM LCTES'09: Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, 2009.

謝辞

本研究では，主任教員である多田好克先生をはじめとした基盤ソフトウェア学講座の皆様のご助力をいただきました．特に，ご多忙にもかかわらず何度とお時間をいただき，研究の相談に乗っていただいた多田先生，小宮先生に改めまして謝意を表します.

また，多田研究室をはじめとした基盤ソフトウェア学講座の学生諸君にも研究方針や実験等に関して多くの意見を頂き，参考にさせていただきました．ここに改めて謝意を表します.