卒業論文

題目

組み込みプロセッサ用小規模HTMの有効性を測るベンチマークの作成と性能解析

指導教員

佐々木敬泰

2017年

三重大学 工学部 情報工学科
コンピュータアーキテクチャ研究室

堀口裕介 (412852)

内容梗概

現在，共有メモリ型マルチコアプロセッサが広く普及しており，並列処理に利用されている．また，一般に並列プログラムの実行順序が非決定的であるため，共有メモリは並行する複数プロセスの実行順序に依存しないデータの一貫性を保持する必要がある．多くのプロセッサでは，その方法として，共有資源をアクセス権で制御するロック手法（以下，単に「ロック」と呼ぶ）が採用されている．しかし，ロックではオーバヘッドが大きく，細粒度ロックの場合，プログラムの複雑化を招く．また，粗粒度ロックの場合，処理に無関係な共有変数に対する操作も逐次実行となるため性能低下の問題がある．そこで，並列に処理を行うために投機的なメモリアクセスを許すトランザクショナルメモリ (Transactional Memory:TM) と呼ばれる手法が提案されている．TM とは，共有資源内のデータに対する読込み，書込みといった一連の処理をトランザクションと定義し，処理する手法である．複数のトランザクションでアクセス先の衝突がなければ並列的に処理を継続するが，衝突がある場合は，トランザクションを中断し，再実行する．特にハードウェアにより実現される TM は，いくつかの商用プロセッサにおいても採用され，ハードウェア TM と呼ばれる（以下，単に「HTM」と呼ぶ）．しかし，それらのプロセッサでは，トランザクション処理を担う回路の規模が大きい．そのため組込み分野での利用が困難である．そこで，当研究室では，扱うトランザクションサイズを限定し，TM をコア内で留め，組込みプロセッサ等の小規模なシステム向けの HTM の実装手法を提案する．しかし実装手法では組み込み向けの小規模な HTM のため，トランザクションサイズの制限とネストトランザクションの禁止という制約を設けている．そのため，従来の TM 専用のベンチマークや並列処理専用のベンチマークをそのまま流用することがでいない．そこで本研究では組込み機器を想定した小規模ベンチマークプログラムを作成し，性能評価，性能解析を行った．結果として実装手法では最大 57%の実行サイクル数の削減を確認した．また性能解析の面では，競合検出成功率が低いことが確認でき，その結果，実行時のスレッド数が増加していくにつれて性能が落ちていく傾向が見られた．しかし今回評価を行った 32 スレッドまででではロックより性能が悪化しているケースは発生しなかった．したがって性能面において実装手法は作成したベンチマーク上では，ロックより有効であること，競合検出手法の改善によってさらなる性能向上が見込めることを確認した．

Abstract

Shared memory multicore processors are widely used in parallel processing. Also, since the execution order of parallel programs is generally non-deterministic, it is necessary for the shared memory to maintain consistency of data that does not depend on the execution order. In many processors, "lock" is adopted as a method. However, overhead is large in lock, and in fine grain lock, the program is complicated. Also, in coarse grain lock, operations on shared variables unrelated to processing are also sequentially executed, so there is a problem of performance degradation. Therefore, transactional memory (TM) has been proposed that allows speculative memory access in order to perform parallel processing. TM performs processing on a transaction basis. A transaction refers to a series of processing such as reading and writing to data in a shared resource. If there are no collisions of access destinations in multiple transactions, processing continues in parallel, but if collision is occured, the transaction is suspended and reexecuted. In particular, the TM realized by hardware is adopted in some commercial processors and is called hardware TM. However, in those processors, the scale of the circuit for transaction processing is large. Therefore, it is difficult to use in the embedded systems. So, we propose a hardware TM implementation method for small scale systems. However, it imposes restrictions on limiting transaction size and ansaporting nested transactions because the implementation method is for embedded applications. Therefore, conventional TM's benchmark and parallel processing's benchmark can not be diverted. My works are creation benchmark, performance evaluation and performance analysis. As a result, the implementation method confirmed reduction of the maximum number of execution cycles by 57%. In terms of performance analysis, it was confirmed that the conflict detection success rate was low, and the performance tended to decrease as the number of threads increased. However, in the evaluation up to 32 threads, there was no case where the performance was worse than the lock. Therefore, we confirmed that on the performance aspect, the implementation method is more effective than the lock on the created benchmark, and further improvements in performance can be expected by improving the conflict detection method.

目 次

図 目 次

表 目 次

1　はじめに

1.1　研究背景

現在，並列処理の需要から共有メモリ型マルチコアプロセッサが広く普及している．並列処理において共有データへの操作を行う際，データの一貫性を保持するために排他制御が必要になる．現在の多くのプロセッサの排他制御にはロックが採用されている．ロックとは共有メモリに対し，操作の前にロックを獲得し，操作の完了後にロックを解放することで，一つのプロセスが排他的に共有メモリに対する変更操作を行う手法である．ロックでは1つのプロセスがロックを獲得している場合，そのプロセス内で扱っている共有メモリに対して，他のプロセスでは操作することができず，待ち状態になる．またロックでは競合が発生する可能性がある箇所すべてにロックをかける必要があり，実際は競合が発生していない場合でもロックをかけて逐次実行しているためにプログラムのオーバーヘッドが大きくなる．

またロックを細かくかけるとプログラムが複雑になったりデッドロックが発生したりする可能性があり，粗にロックをかけると無関係の共有変数も逐次実行になり性能が著しく劣化するため慎重な設計が要求される．このようなロックの問題点に対して提案された排他制御がトランザ

クショナルメモリ (TM) である．TM とは実際に競合が発生した場合に片方の処理を中断し再実行を行うことで排他制御を実現している手法である．そのためロックをかける必要がなく，ロックの粒度に対する問題や，デッドロックを回避できる．TM の実装手法はソフトウェアに実装する方法（STM）とハードウェアに実装する方法（HTM）の 2 通り存在する．現状の HTM の問題点として，トランザクション処理を完全にサポートさせるとハードウェア規模が大きくなるという問題がある。そこで，当研究室では扱うトランザクションサイズに制限を掛け，コア内部のストアバッファを利用することで TM をコア内で留める組み込み用プロセッサ等の小規模なシステム向けの HTM の実装手法が提案している [2]．しかし TM は新しい技術であり，性能評価を行うためのベンチマークプログラムがあまり存在しない．さらに本研究で扱う HTM は小規模なため，トランザクションサイズに制約があり、既存のベンチマークプログラムを用いて評価することは困難である．そこで本研究では，文献 [2] にて提案されている小規模 HTM の有効性を示すため，小規模 HTM の性能解析できるベンチマークを作成し，その結果を基に小規模 HTM の有効性を考察する．

以降，本論文は次の様に構成される．まず，続く章では排他制御の方法

の概要としてロック，トランザクションのそれぞれについて説明する第 3 章では，小規模 HTM の実装手法についての紹介．第 4 章，第 5 章では，問題点として現状小規模 HTM の有効性を測るものがないことについての説明と TM 用ベンチマークの関連研究について説明する．ベンチマークプログラムの詳細と評価結果については第 6 章で述べる．その結果を基に第 7 章にて小規模 HTM の有効性を考察する．

2　排他制御の方法の概要

ロックとトランザクションのそれぞれについて概要を述べる

2.1　ロック

ロックは一般的な排他制御の手法である．この方法は共有メモリに対する操作の前にロックを獲得し，操作の完了後にロックを解放することで，一つのプロセスが排他的に共有メモリに対する変更操作を行う手法である．ロックは Pthread や OpenMP などの API がサポートしている．ロックでは，スレッドで競合の発生が予想される場合には常にロックを確保している．しかし，実際に競合の発生頻度は低い場合，不必要なオーバーヘッドが生じる問題がある．また，ロックを細かくかけるとプログラムが複雑になったりデッドロックが発生したりする可能性があり，粗にロックをかけると無関係の共有変数も逐次実行になり性能が著しく劣化するため慎重な設計が要求される．デッドロックとは 2 つ以上のスレッドあるいはプロセスなどの処理単位が互いの処理終了を待ち，結果としてどの処理も先に進めなくなることをいう．このようなロックの問題に対して提案された制御手法が TM である．

2.2 トランザクショナルメモリ

TM とは，データを読み込み，変更し，書き込むという一連の操作を一つのトランザクションと見立てて実行する．共有メモリへの操作で競合が発生した時は片方のプロセスをトランザクションの直前に戻し，再実行することで，データの一貫性を保持している．したがって，共有メモリへのロックを必要としないので，ロックの問題点である不要なオーバーヘッドの発生やデッドロックの問題を回避することができる．

トランザクション処理の例を図 2.1 を用いて説明する．今，二つのコアのそれぞれで Thread1，Thread2 が実行されていると仮定する．それぞれのスレッドはトランザクション Tx.A，Tx.B を実行している．時刻 t0，t2 において，互いのトランザクションがアドレス a からロードを行っている．そして，時刻 t3 では，Tx.A がアドレス a に対してストアを試みるが，Tx.B でのアクセスがあったことを確認し，トランザクションアボートが起きる．その後，時刻 t4 にて競合対象であった Tx.A がアボートしたため，Tx.B がアドレス a へ書き込みができるので Commit される．その後，Tx.A は，再実行が行われる． では TM の実装手法であるソフトウェアトランザクショナルメモリ（STM）とハードウェアトランザクショナルメモリ（HTM）について述べる．

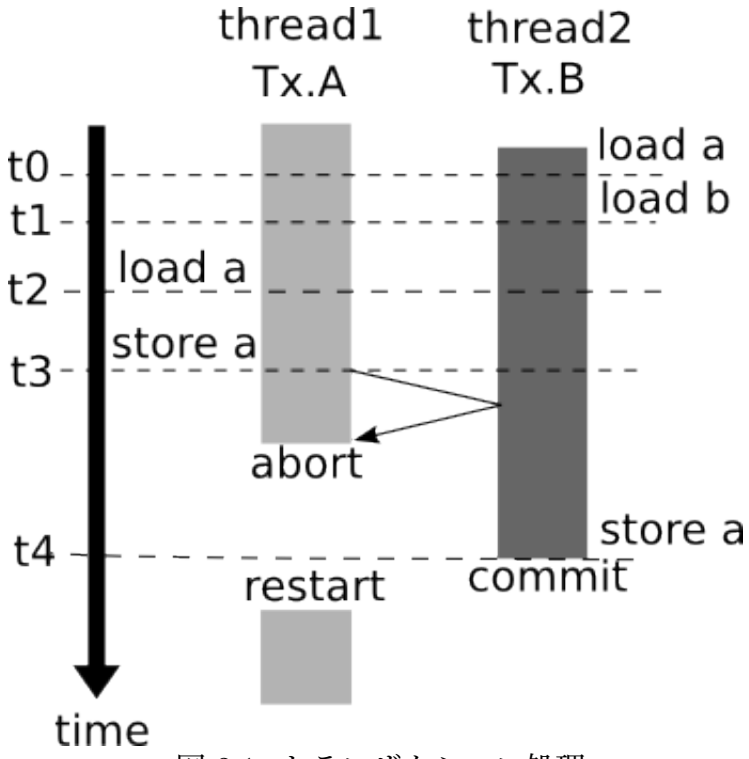

図 2.1: トランザクション処理

2.2.1 STM

ソフトウェアトランザクショナルメモリ（STM）はTMのアイデアをソフトウェアのみで行う方法である．STMではTM用の特殊なハードウェア機構を用いることなくトランザクショナルメモリを実現できる一方で，ソフトウェアを用いて処理をするため，生じるオーバーヘッドが大きくなるという問題点がある．STMをサポートする有名なモデルとしてClojure[3]が存在する．Clojureとは，JAVA仮想マシン上で動く関数型言語である．STMはロックに比べプログラムの設計自体は容易になっているが，命令セットはTM専用の命令をサポートしていないため，トランザクション間でwrite命令が頻発すると実行速度が落ちるという問題がある．

2.2.2 HTM

ハードウェアトランザクショナルメモリ（HTM）はTMをハードウェアで実装するため，オーバーヘッドを最小限にとどめることができる．しかし，トランザクショナルメモリの機構を実現するハードウェアの実装コストおよび，ハードウェア規模が膨大になる問題がある．HTMをサポートする有名なモデルとしてIntel社のTSX[4]が存在する．TSXでは競合

が発生するまではロックをかけずに命令を実行しているが競合が発生した際に再実行する命令にロックをかけて行うようなコーディングの仕方が勧められている．

3　実装手法

前章のHTMの問題に対し当研究室では，扱うトランザクションサイズに制限を掛け，コア内部の既存資源を利用することでトランザクショナルメモリをコア内で留める組込み用プロセッサ等の小規模なシステム向けのHTM実装手法（参考文献）を提案している．この実装手法では従来のHTMに比べ，ハードウェア規模，実装コスト，および消費電力の低減が確認されている．しかしこの実装手法は組み込み用の小規模なHTMの実装手法のため，扱うトランザクションサイズに厳しい制限を掛けている．またそのことに関係してネストトランザクションを禁止している[1]．ネストトランザクションとはトランザクション処理の中にトランザクション処理が含まれている階層的な構造をもつトランザクションである．

[1]ただし，トランザクションサイズの制限は小規模HTMに限らず，HTM全般に言えることである．

4　関連研究

当研究室で提案されている小規模 HTM について評価を行う際，TM のベンチマークの論文の 1 つとして STAMP（Stanford Transactional Applications for Multi-Processing）[1] がある．STAMP ではトランザクション処理が頻繁なものから希なものまで広いケースをカバーしている．また STAMP は，ハードウェア，ソフトウェア，およびハイブリッドシステムを含む TM システムの多くのタイプで移植可能である．TM 全般に使用できるベンチマークベンチマークである．また並列処理用のベンチマークとして SPLASH-2[5]，姫野ベンチマーク [6]，組み込み用の小規模なベンチマークとして MiBench[7] が挙げられるが次節で述べる問題より流用できない．

5　問題点

評価対象の HTM には第 3 章で述べたように，扱うトランザクションサイズに厳しい制限を掛けている．そのため前節で紹介したベンチマークをそのまま流用することができないという問題がある．また評価対象の HTM について，既存のベンチマークでは，動作解析を行うことができず，性能劣化が発生していた場合の原因等の確認ができないので，改善点が

考察しにくいという問題がある．

6　提案手法

本章では前章で述べた問題点を解決するために，一般的な並列実行モデルに基づいたベンチマークプログラムを設計し，性能分析を行う．

6.1　提案手法の実行モデル

並列プログラムの実行モデルとしては，様々なものが存在するが，本研究では図 6.2 に示すようなマスタースレーブ型の実行モデルを採用する．このモデルの採用理由として，評価対象の HTM の性能について現状ほとんど情報がないため，ベンチマークを走らせたとき，どのスレッドで競合が発生しているか等の，動作解析が容易になるため，このモデルを採用した．また小規模 HTM では，トランザクションサイズの制約が厳し

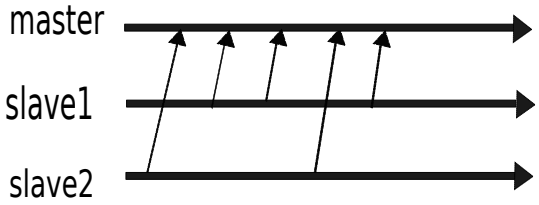

図 6.2: データフロー

くトランザクション間に存在する命令数に注意する必要がある．例えばトランザクション間で関数呼び出しをすると呼び出し先で変数を多量に

使う場合，TM 用に用意したストアバッファの容量を超えたり，ループ処理が何度も競合し，無限ループに陥ってしまうことで，最終的にプログラムが走らなくなることもある提案手法では実際に共有資源を操作する前後でのみ排他制御をし，トランザクションサイズを抑えたもので，なおかつネストトランザクションが存在しないモデルを想定した．

6.2 評価環境

評価環境は表 6.1 の通りである．

トランザクションサイズ小，中，大とは例えば，トランザクションサイズ小の場合，図 6.3 のようにトランザクション専用命令間に含まれる命令が 1 つで実行サイクル数が数命令程度のもの．トランザクションサイズ中は，専用命令間に含まれる命令が 5 命令程度のも．トランザクションサイズ大は，専用命令間に含まれる命令が 10 命令程度のものとして定義し

表 6.1: 評価方法

評価項目	実行サイクル数
評価環境	ソフトウェアシミュレータ
比較対象	Pthread ロック
変更したパラメータ	
トランザクションサイズ	小・中・大
競合確率	1% 〜 100%
スレッド数	2・4・8・16・32

```
xbegin();
shared_data++;
xend();
```

図 6.3: 擬似コード

ている．次節では実際に評価のために作成した提案手法について述べる．

6.3 評価用プログラムについて

提案手法では，常に共有資源に対して計算処理をしているスレッドを 1 つ作成し，他のスレッドは，ランダムなタイミングで共有資源への計算処理を行うものを作成し，この一連の処理を for 文で 10 万回実行した時のステップ数を評価の対象とした．図 6.4，図 6.5 は作成したベンチマークのフローチャートである．図 6.4 はランダムなタイミングで計算処理を行うスレッドでこのスレッドが作成したモデルのスレーブに該当するスレッドである．図 6.5 は扱っているすべての変数に順番に計算処理を行うスレッドでマスターに該当するスレッドである．それぞれのスレッドで扱っている shared_data を競合が発生する可能性のある変数として用意している．ベンチマークでは図 6.4 のスレーブに該当するスレッドを複数と，図 6.5 のマスターに該当するスレッドを 1 つ用意した．なお今回作成したモデルではスレーブ間での競合は発生していない．評価をとるに当

たって競合確率，スレッド数，トランザクションサイズをパラメータとした．競合確率は1%から100%までを5%刻みに設定，スレッド数は2，4，8，16，32の5パターン，トランザクションサイズは前節の評価環境で述べた通りに定義した3 パターンでありそれぞれのパターンを組み合わせて実行したときのサイクル数を評価の対象とした．パラメータ値の競合確率とは実際に競合が発生した確率ではなく，競合が発生する可能性があり，排他制御の必要なケースが発生する確率である．具体的には図6.4のif 文の返り値が真になり共有変数に対しての書き込みが発生する確率である．プログラムの動作としては，例えば2スレッドの場合は1つのスレッドは常に共有変数に対して計算処理をしていて，2つ目のスレッドはランダムなタイミングで共有変数に対して計算処理を行うものを作成した．4スレッドの場合は3つのスレッドがそれぞれに別の変数に対してランダムなタイミングで計算処理を行い，1つのスレッドが先の3つのスレッドで扱っている全ての変数に対して順番に計算処理を行うものを作成した．スレッド数が8以降に関しても同様で，この一連の処理を10万回ループさせた時の実行サイクル数を性能評価の項目として用いた．

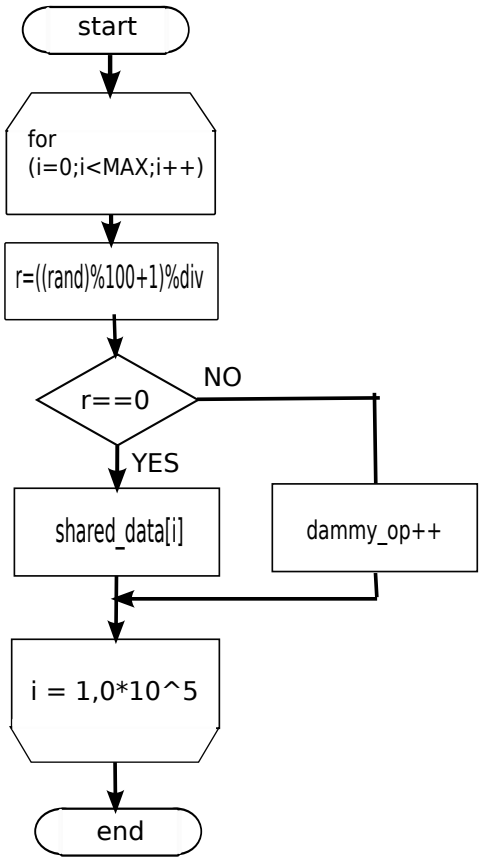

図 6.4: ランダムなタイミングで計算処理を行うスレッド

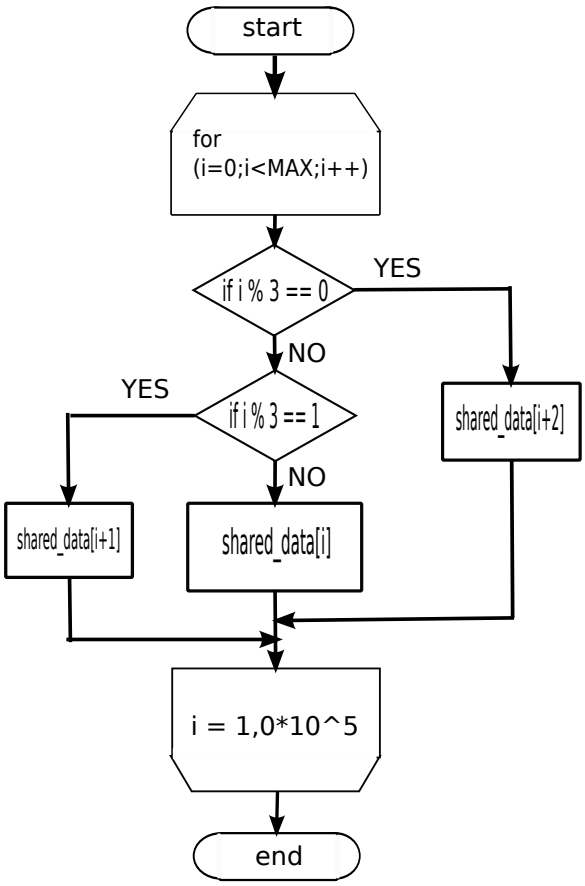

図 6.5: 扱っているすべての変数に順番に計算処理を行うスレッド

6.4 評価結果

図 6.6 から図 6.8 が作成したベンチマークを用い，評価をとった際に得られた結果をグラフにまとめたものである．グラフの X 軸は競合確率，Y 軸はロックのサイクル数を 1 として正規化したときの実装手法の実行サイクル比である．

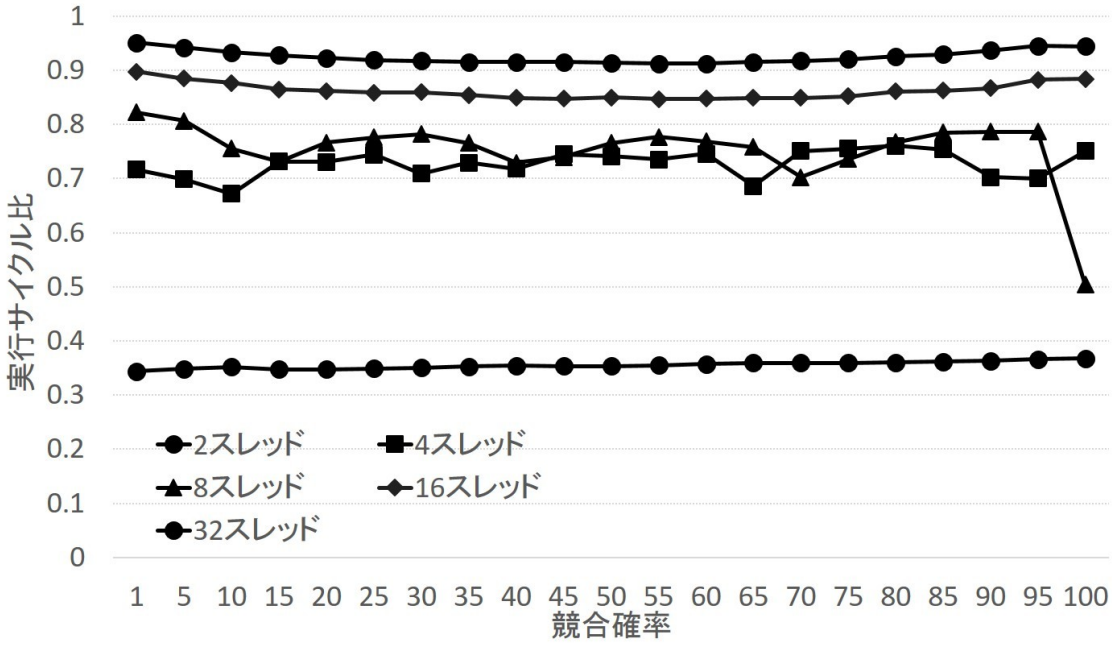

図 6.6: トランザクションサイズ小

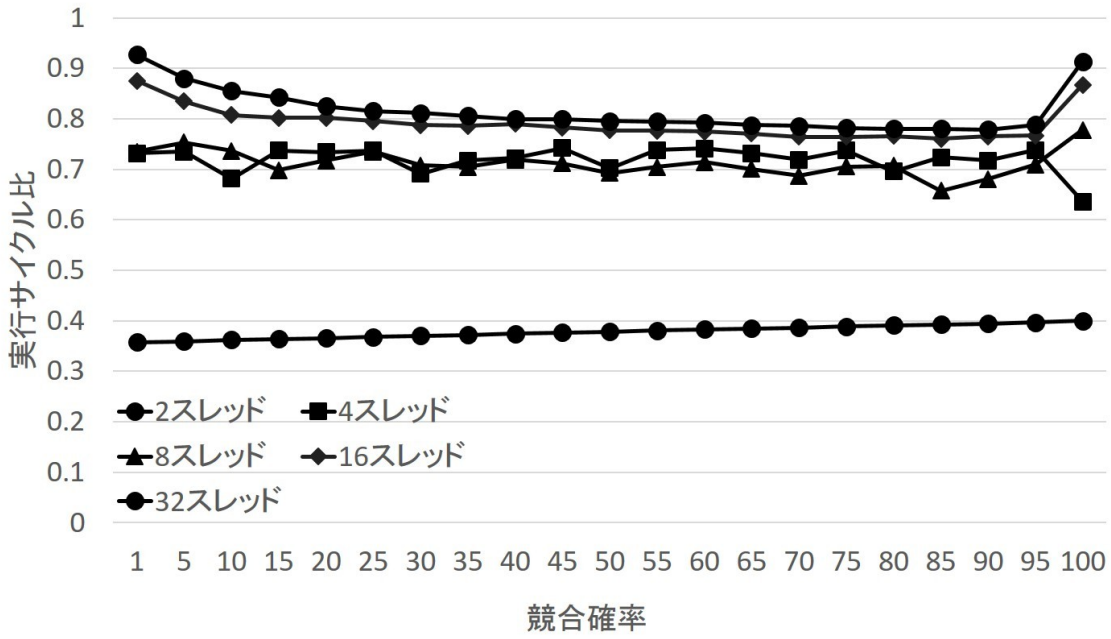

図 6.7: トランザクションサイズ中

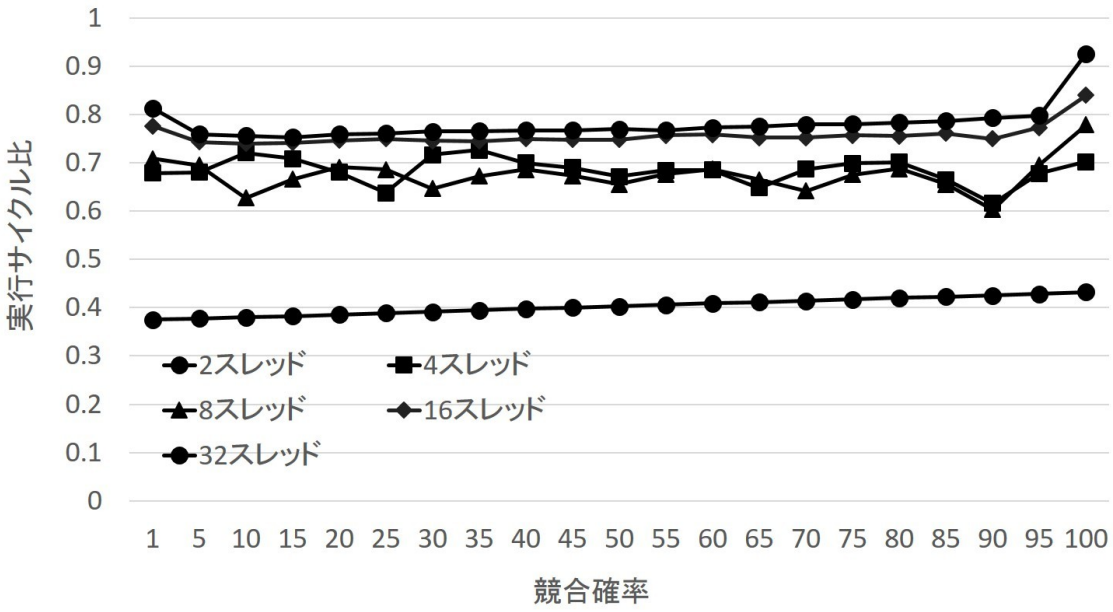

図 6.8: トランザクションサイズ大

以上の結果ではトランザクションサイズ小で，スレッド数 2 の時ロックに比べて実行ステップ数が最大で 65%削減できていることが確認できた．またトランザクションサイズの全てのパターンについてロックより性能が悪化しているケースは確認できなかった．そのため，今回作成したような小規模なマスタスレーブ型のモデルについてはロックより有効な手法だと言うことが確認できた．評価結果の傾向としてスレッド数が増加していくにつれてロックとの性能差が縮小していく傾向が見られた．この原因としては競合検出手法にあると考察している．評価対象の HTM では競合検出の際，アクセスアドレスに対し，ハッシュ関数を用いて変換した値を 32 ビットレジスタに格納し，その値を利用している．そのためアクセスアドレスが異なっていて実際に競合が発生していない場合でもハッシュ関数により変換された値が同一の場合，競合検出されロールバックが発生する．この競合検出の失敗率が約 90%程度ということが確認でき，スレッド数増加により失敗率も増加していることが確認できた．

7　おわりに

評価の結果，今回のベンチマークのように小規模なプログラムで競合が発生する可能性がある場合評価対象の HTM は有効であることが確認できた．しかし競合検出精度が低く不必要な再実行が発生していることも確認できた．今後の課題として，現状競合検出手法で用いられるハッシュ関数を改良し，異なるアクセスアドレスを変換する際，変換後の値が衝突しないハッシュ関数を考案し，検出精度を上昇させる改良が必要であると考察している．また，ハッシュ関数を改良後，競合検出精度の上昇が確認できた場合，フォークジョインモデルのベンチマークを用いての性能評価も行う必要があると考察している．

謝辞

本研究を行うにあたり，多数のご指導を頂きました近藤利夫教授，佐々木敬泰助教，並びに深澤研究員に深く感謝いたします．また，コンピュータアーキテクチャ研究室の学生には常に刺激的な議論を頂き，精神的にも支えられました．

参考文献

[1] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis and Kunle Oluktun.STAMP:Stanford Transactional Applications for Multi-Processing In Proceeding of The IEEE International on Workload Characterization September 2008.

[2] Takahiro Sakurada Keisuke Sasaki Yuki Fukazawa Tosio Kondo Efficient implementation method of a compact HTM into processor cores. Swopp 2016 August.

[3] Stuart Halloway(著), 河合志朗 (訳), プログラミング Clojure, 株式会社オーム社 ,2010.

[4] Intel. Intel 64 and IA-32 Architectures Software Developer 's Manual, June 2013.

[5] The Modified SPLASH-2, <http://www.capsl.udel.edu/splash/> (2017 年 3 月 6 日アクセス)

[6] 姫野龍太郎, 姫野ベンチマーク, <http://accc.riken.jp/supercom/himenobmt/> (2017 年 3 月 6 日アクセス)

[7] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, Richard B. Brown, MiBench: A free, commercially representative embedded benchmark suite, <http://vhosts.eecs.umich.edu/mibench/Publications/MiBench.pdf/> (2017年3月6日アクセス)

A　プログラムリスト

```
#define ALIGNMENT 64
#define SKTX
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sched.h>
#include <pthread.h>
#ifdef __WIN32
#include<intrin.h>
#endif
#ifdef TSX
#include <immintrin.h>
#endif
#include <time.h>
#define MAX 100000
/*実行時./prog スレッド数 競合確率 トランザクションサイズ 乱数の
seed値　をパラメータの設定時にそれぞれ変更し実行*/
typedef struct{
  int thread_num;
  int thread_id;
  int conflict_rate;
  int size;
}arg_t;
pthread_mutex_t mutex;
int set_size(char *size);
void master(arg_t *arg);
void slave(arg_t *arg);
void slave_small(const int thread_id,const int conflict_rate);
void slave_middle(const int thread_id,const int conflict_rate);
void slave_large(const int thread_id,const int conflict_rate);
int shared_data[100];
//int GetRandom(int min,int max);
int main(int argc, char *argv[]){
  if(argc != 5){
    printf("usage:./prog thread_num conflict_rate size seed\n");
    exit(-1);
  }
  const int thread_num = atoi(argv[1]);
  const int slave_thread_num = thread_num -1;
  const int conflict_rate = atoi(argv[2]);
  const int size = set_size(argv[3]);
  const int seed = atoi(argv[4]);
  if(size == -1){
    printf("error: size is wrong \n");
    exit(-1);
  }
  pthread_t master_thread;
  pthread_t *slave_thread;
  arg_t *arg = (arg_t*)malloc(sizeof(arg_t)*thread_num);
  int i;
  for(i = 0;i < thread_num;i++){
    arg[i].thread_num = thread_num;
```

```
    arg[i].conflict_rate = conflict_rate;
    arg[i].thread_id = i;
    arg[i].size = size;
  }
  srand(seed);
  slave_thread = (pthread_t*) malloc(sizeof(pthread_t)*(slave_thread_num));
  pthread_create(&master_thread,NULL,(void *)master,(void*)&arg[0]);
  for(i = 0;i < slave_thread_num;i++){
    pthread_create(&slave_thread[i],NULL,(void *)slave,(void*)&arg[i+1]);
  }
  for(i = 0;i < slave_thread_num;i++){
    pthread_join(slave_thread[i],NULL);
  }
  pthread_join(master_thread,NULL);
  return 0;
}
void master(arg_t *arg){
  int i;
  int r;
  const int thread_num = arg->thread_num;
#ifdef SKTX
  for(i=0; i < MAX; i++){
    r = i % thread_num;
    __asm__(".word 0x49200001");//TM_Begin
    shared_data[r]++;
    __asm__(".word 0x49200002");//TM_End
    __asm__(".word 0x00000000");//NOP
  }
#else
   for(i=0; i < MAX; i++){
     r = i % thread_num;
     pthread_mutex_lock(&mutex);
     shared_data[r]++;
     pthread_mutex_unlock(&mutex);
   }
#endif
}
#define SMALL  1
#define MIDDLE 2
#define LARGE  3
void slave(arg_t *arg){
  const int thread_id = arg->thread_id;
  const int conflict_rate = arg->conflict_rate;
  switch(arg->size){
  case SMALL: slave_small(thread_id,conflict_rate);break;
  case MIDDLE: slave_middle(thread_id,conflict_rate);break;
  case LARGE: slave_large(thread_id,conflict_rate);break;
  default: break;
  }
}
int set_size(char *size)
{
  if(!strcmp(size,"XS") || !strcmp(size,"xs")){
```

```
    return SMALL;
  }
  if(!strcmp(size,"S") || !strcmp(size,"s")){
    return MIDDLE;
  }
  if(!strcmp(size,"M") || !strcmp(size,"m")){
    return LARGE;
  }
  return -1;
}
void slave_small(const int thread_id,const int conflict_rate){
  int i;
  int r;
  int dummy_op = 0;
#ifdef SKTX
  for(i=0;i<MAX;i++){
     r = rand()%100;
     if(r < conflict_rate){
       __asm__(".word 0x49200001");//TM_Begin
       shared_data[thread_id]++;
       __asm__(".word 0x49200002");//TM_End
       __asm__(".word 0x00000000");//NOP
     }
     else{
       dummy_op++;
     }
  }
#else
  for(i = 0; i < MAX; i++){
    r = rand()%100;
    pthread_mutex_lock(&mutex);
    if(r < conflict_rate){
      shared_data[thread_id]++;
    }
    else{
      dummy_op++;
    }
    pthread_mutex_unlock(&mutex);
  }
#endif
};
void slave_middle(const int thread_id,const int conflict_rate){
  int i;
  int r;
  int dummy_op = 0;
#ifdef SKTX
  for(i=0;i<MAX;i++){
     r = rand()%100;
     if(r < conflict_rate){
       __asm__(".word 0x49200001");//TM_Begin
       shared_data[thread_id]++;
       shared_data[thread_id]++;
       dummy_op++;
       dummy_op++;
       dummy_op++;
```

```
        __asm__(".word 0x49200002");//TM_End
        __asm__(".word 0x00000000");//NOP
      }
      else{
        dummy_op++;
        dummy_op++;
        dummy_op++;
        dummy_op++;
        dummy_op++;
      }
  }
#else
  for(i = 0; i < MAX; i++){
    r = rand()%100;
    pthread_mutex_lock(&mutex);
    if(r < conflict_rate){
      shared_data[thread_id]++;
      shared_data[thread_id]++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
    }
    else{
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
    }
    pthread_mutex_unlock(&mutex);
  }
#endif
};
void slave_large(const int thread_id,const int conflict_rate){
  int i;
  int r;
  int dummy_op = 0;
#ifdef SKTX
  for(i=0;i<MAX;i++){
     r = rand()%100;
     if(r < conflict_rate){
       __asm__(".word 0x49200001");//TM_Begin
       shared_data[thread_id]++;
       shared_data[thread_id]++;
       dummy_op++;
       dummy_op++;
       dummy_op++;
       dummy_op++;
       dummy_op++;
       dummy_op++;
       dummy_op++;
       dummy_op++;
       __asm__(".word 0x49200002");//TM_End
       __asm__(".word 0x00000000");//NOP
```

```
     }
     else{
       dummy_op++;
       dummy_op++;
       dummy_op++;
       dummy_op++;
       dummy_op++;
       dummy_op++;
       dummy_op++;
       dummy_op++;
       dummy_op++;
     }
  }
#else
  for(i = 0; i < MAX; i++){
    r = rand()%100;
    pthread_mutex_lock(&mutex);
    if(r < conflict_rate){
      shared_data[thread_id]++;
      shared_data[thread_id]++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
    }
    else{
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
      dummy_op++;
    }
    pthread_mutex_unlock(&mutex);
  }
#endif
};
```