Vol.2018-ARC-230 No.12
Vol.2018-SLDM-183 No.12
Vol.2018-EMB-47 No.12
2018/3/7

最適な並行性制御を適用するコード生成手法の検討

林 昌樹[1]　二間瀬 悠希[1]　多治見 知紀[1]　塩谷 亮太[2]　五島 正裕[3]　津邑 公暁[1]

概要：トランザクショナルメモリは，トランザクションとして定義された区間を投機的に並行実行することで，粗粒度ロックと同等以上の記述性と，細粒度ロックと同等以上の性能とを両立しうるパラダイムとして期待されている．しかし，共有変数に対するアクセス競合が頻発する場合は，必ずしも高い性能を実現し得ない．本稿では，プログラム中に定義されたトランザクションのうちのいくつかを，ロックによる制御同様，排他的に実行した場合，性能にどのような変化があるかを調査・解析する．また，ロックとトランザクショナルメモリを併用することによって発生する一貫性制御の問題を指摘し，これを解決する並行性制御手法を提案する．そして，調査で得られた知見に基づき，プログラマが定義した各トランザクションに対し，提案する並行性制御手法を適用すべきか否かを自動的に判別した上で，その判別結果に沿ったコード生成を行う手法について検討する．

1. はじめに

共有メモリ型の並列プログラミングでは共有メモリに対するアクセス調停の枠組みとしてロックが広く用いられてきた．ロックでは同一のロック変数で保護された区間同士を排他実行することで並行性制御を行う．しかし，ロックを粗粒度に定義した場合，実行時の並列度が低くなり性能低下を招く．一方細粒度に定義した場合，高い並列度を達成し得るが，プログラミングが非常に複雑になる．このため，ロックはプログラマにとって必ずしも利用しやすい仕組みではない．

このロックに代わる並行性制御機構として**トランザクショナルメモリ（Transactional Memory: TM）**[1] が提案されている．ロックを用いる場合にクリティカルセクションとして保護する区間を，TM では**トランザクション（Transaction: Tx）**として定義し，それらを投機的に並行実行することで高い並列度が得られる．この点から，TM は粗粒度ロックと同等の記述性と細粒度ロックと同等以上の性能を両立しうると期待されている．TM では Tx を実行中の複数のスレッド間でアクセス競合が発生することなく Tx の実行が終了した場合，Tx 内で行った変更を確定し，共有メモリ上に反映する（**コミット；Commit**）．一方，競合が発生した場合，一方の Tx の実行状態を破棄する（**アボート；Abort**）ことで，競合を解決し，アボートされた Tx は再実行される．競合発生時に必要となるこれらの処理は，性能に大きな影響を与えうる．また，TM は各スレッドのメモリアクセスを監視し，競合の発生を検出する，**競合検出（Conflict Detection: CD）**機構と，Tx アボート時に Tx 実行開始前の状態を復元するための**バージョン管理（Version Manegement: VM）**機構を備えている必要があり，これらの制御によるオーバヘッドが発生する．

本稿では，プログラムの構造と Tx の定義を解析し，必要に応じて自動的にプログラムを書き換えることで性能向上を目指す．そのために，同一 Tx を排他的に動作させることで競合を抑制できる場合があることに着目し，さまざまな Tx に対し，投機実行した場合と排他実行した場合の実行結果を比較する．そしてその調査結果から，排他実行することで性能向上する Tx の特徴を明らかにし，Tx として定義された各区間に最適な並行性制御を適用したコードを生成する手法について検討する．また，TM とロックとの併用によって発生する問題について指摘し，これを解決する手法を提案する．

2. 同一 Tx どうしの排他実行とその性能調査

排他実行することで性能が向上する Tx の特徴を解析することを目的とし，さまざまな Tx を排他実行した場合の性能変化を調査した．

2.1 調査環境

調査環境を**表 1** に示す．なお，コンパイラは GCC 5.4.0

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 名古屋大学
Nagoya University
[3] 国立情報学研究所
National Institute of Informatics

Vol.2018-ARC-230 No.12
Vol.2018-SLDM-183 No.12
Vol.2018-EMB-47 No.12
2018/3/7

表 1　調査環境

OS	ubuntu16.0.4
CPU	Intel Core i7-8700K
clock	3.70GHz
physical/logical #cores	6/12
L1d/L2d/L3d cache	32KB/256KB/12288KB

を用い，コンパイルオプションとして-O3 を指定した．また，調査対象のプログラムとして STAMP[2] から Kmeans-high，Genome，Intruder の 3 つを使用し，実行時オプションはすべてデフォルトの値を用いた．スレッド数に応じて性能が異なると考えられるため，1，2，4，8，16 スレッドでそれぞれ実行した．また，同一 Tx 同士を排他制御した場合の調査には，当該 Tx の開始と終了をそれぞれロックの獲得と解放に変換した実装を用いたが，この実装では，実行可能なプログラムが限定される．この原因と解決方法については 3 章で説明する．調査では，HTM として Intel 社のプロセッサに実装されている TSX[3], [4] の RTM を，STM として TL2[5] を使用した．なお，RTM では forward progress を保証するため，先行研究 [6][7] にならい，一度の実行で競合を繰り返した Tx はロックで保護する実行に切り替わる実装を採用した．なお，プログラムはそれぞれ 10 回ずつ実行し，その平均を評価結果として示す．

2.2　調査結果と考察

HTM 及び STM で，ベンチマークプログラム中の各 Tx を，投機実行または排他実行した場合のそれぞれの実行時間を**図 2** から**図 4** に示す．また，STM で各 Tx を投機実行または排他実行した場合に発生したアボート数・コミット数を**図 5** から**図 7** に示す．図中の凡例は，Lock，{HTM, STM}-default はそれぞれ，全ての Tx をロック，HTM, STM で実行した場合，{HTM, STM}-Tx.n は，HTM, STM それぞれにおいて，ID が n である Tx のみをロックで排他実行した場合の結果を示している．{HTM, STM}-Tx.n では，Tx.n の開始と終了をロックの獲得と解放にそれぞれ書き換えており，当該 Tx のコミットおよびアボート回数はゼロとなる．よって，全ての Tx の総アボート数と，ある Tx を排他実行した場合のアボート数との差から，当該 Tx によって発生したアボート数が概算できる．最後に，STM において 1 スレッドで実行した場合の，各 Tx の一回当たりの実行時間と，プログラム中で各 Tx が実行された回数とを**表 2** に示す．この表の値は 1 スレッドで実行した場合の結果であり，全ての Tx はアボートされることなく一度の実行でコミットされている．

2.2.1　Kmeans-high

図 2 から Kmeans-high では，HTM，STM 共に Tx.0 を排他実行した場合が性能が最も高いことが分かる．図 5 から，Tx.0 を排他実行することでアボート数が大きく抑制されており，それ以外の Tx を排他実行した場合はあまりアボート数が抑制されていないことが分かる．そして表 2 から，Tx.0 の実行回数が非常に多く，次いで Tx.1 が多いことと，Tx.0 の実行時間が他の Tx と比べて非常に長いことが分かる．なお Kmeans は**図 1** に示すような構造を持ち，Tx.0 が連続的に繰り返し実行される構造となっている．プログラムを調査すると，全ての Tx の中で，Tx.0 が最も多くの共有変数へのアクセスを含んでいた．以上から，Tx.0 は実行回数が多く，Tx 内で多くの共有変数へアクセスするため，アボートを頻発させる Tx であると考えられる．よって，{HTM, STM}-Tx.0 は Tx.0 を排他実行することで，アボート数が大きく抑制されたため高い性能となったと考えられる．一方で Tx.1 および Tx.2 は共有変数へのアクセス回数が非常に少ない．また，Tx.0 の実行時間が非常に長いことと，その Tx.0 が連続的に繰り返し実行されるプログラム構造から，あるスレッドが Tx.1，Tx.2 を実行する時，他のスレッドは Tx.0 を実行している可能性が高く，Tx.1 と Tx.2 同士は並行実行されにくいと考えられる．よって，これらの Tx を投機実行した場合でも，アボートされることは少ないと考えられるため，{HTM, STM}-Tx.{1,2} は，排他実行により並列度が低下しただけで，性能が伸びなかったと考えられる．

以上から，Kmeans のような構造を持つプログラムでは，共有変数へのアクセス回数が多く連続的に繰り返し実行される Tx は排他実行した方がよく，そうでない Tx は投機実行した方が良いと考えられる．

2.2.2　Genome

図 6 から分かるように，Genome はアボートの発生が少

表 2　各プログラム内で定義された各 Tx の実行時間と実行回数

プログラム名	TxID	実行時間（ms）	実行回数
Kmeans-high	Tx.0	1.5011	7,405,568
	Tx.1	0.2713	2,468,485
	Tx.2	0.4576	113
Genome	Tx.0	6.1704	1,398,102
	Tx.1	0.3977	16,321
	Tx.2	5.4205	1,028,223
	Tx.3	0.5527	16,321
	Tx.4	0.4720	30,251
Intruder	Tx.0	0.1619	7,809,376
	Tx.1	3.6625	7,809,375
	Tx.2	0.1345	7,809,375

```
1  while(...){
2    while(...){
3      for(...){
4        ---Tx.0---
5      }
6      ---Tx.1---
7    }
8    ---Tx.2---
9  }
```

図 1　Kmeans のプログラム構造

Vol.2018-ARC-230 No.12
Vol.2018-SLDM-183 No.12
Vol.2018-EMB-47 No.12
2018/3/7

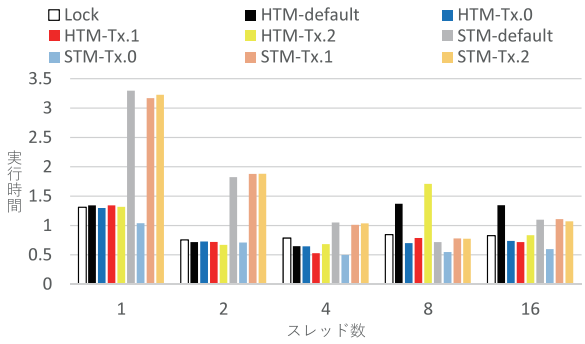

図 2　Kmeans-high 実行時間

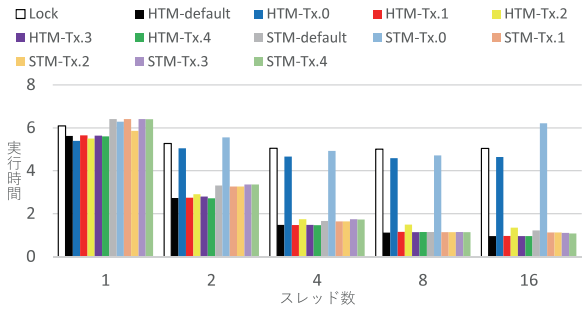

図 3　Genome 実行時間

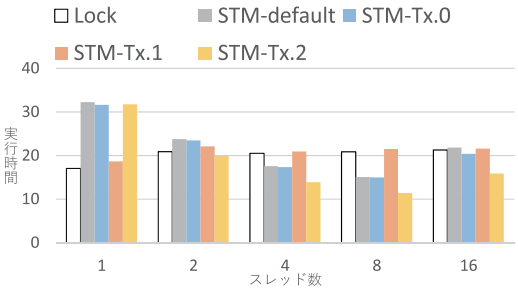

図 4　Intruder 実行時間

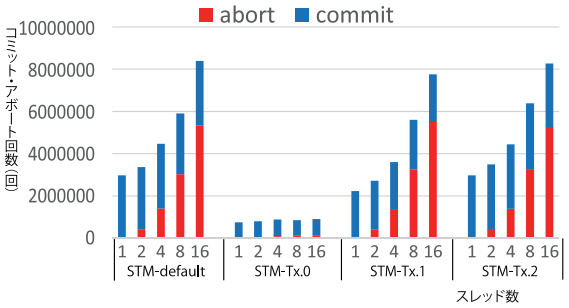

図 5　Kmeans-high コミット・アボート数

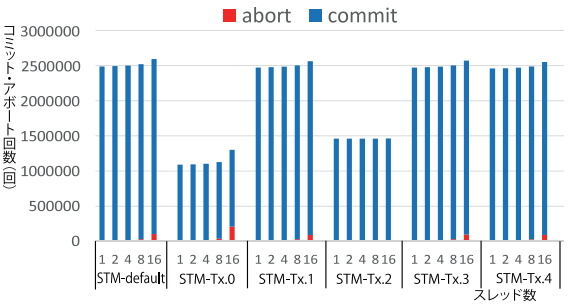

図 6　Genome コミット・アボート数

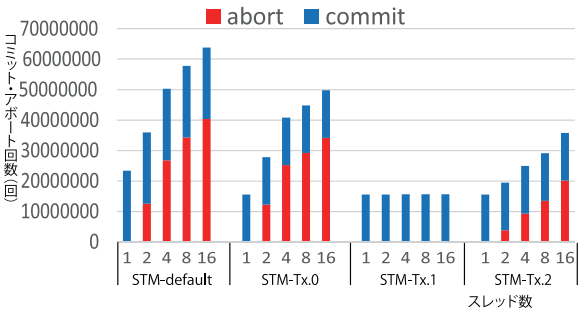

図 7　Intruder コミット・アボート数

ないプログラムであるが，図 3 に示すように Tx.0 を排他実行した場合に性能が低く，それ以外のモデルではほぼ性能差がない．

まず Tx.{1,3,4} は実行回数が少ないため，性能に与える影響が小さく，モデル間に性能差がなかった．{HTM, STM}-Tx.2 については排他制御により，アボート数の抑制と並列度の低下それぞれが性能に与える影響が相殺されたことで，全体性能に変化が見られなかったと考えられる．一方 Tx.0 は，**図 8** に示す Genome のプログラム構造から分かるように，Kmeans の Tx.0 同様に for 文で連続的に繰り返し実行される Tx で，共有変数へのアクセス回数も多いため，排他実行に向くと考えられる．しかし，当該 for 文の直後にバリア同期が設けられており，Tx.0 とバリア同期以降の Tx が並行実行されない構造になっている．Tx.0 は実行時間が長い Tx であるため，排他実行されることでバリア同期までの実行時間が非常に長くなり，またバリア同期のために他 Tx とも並行実行されないことから，性能が著しく低くなったと考えられる．

以上のことから，連続的に繰り返し実行される Tx であっても，その実行時間が長く，他 Tx との並行実行がされにくい Tx は，排他実行に向かないと考えられる．

2.2.3　Intruder

図 4 の結果から，Intruder は，排他実行する Tx によって性能が異なる．プログラムは図 11 に示すような構造をしており，Tx.0〜Tx.2 が順に繰り返し実行される．STM-Tx.2 は STM-default よりも性能が高くなっている．ここで，表 2 より，Tx.2 は実行時間が短く，Tx.1 は長いことが分かる．よって， Tx.2 同士を排他実行した場合でも，Tx.1 などの長く Tx.2 と競合しにくい Tx と並行実行されやすいため，並列度が大きく損われず，Tx.2 同士の競合発生回数を抑えたことで性能が向上したと考えられる．一方 Tx.1 は実行時間が長いため，排他実行することで総実行時間が大きく増大してしまい，Tx.1 同士の競合に伴うオーバヘッドを削減したことにより得られた恩恵をこれが上回ったことで，STM-Tx.1 は性能が低くなったと考えられる．Tx.0 については，図 7 の結果を見ると，STM-Tx.0 のアボート

Vol.2018-ARC-230 No.12
Vol.2018-SLDM-183 No.12
Vol.2018-EMB-47 No.12
2018/3/7

```
1    for(...){
2      ---Tx.0---
3    }
4    Barrier Synchronization
5    for(...){
6      while(...){
7        ---Tx.1---
8      }
9      for(....){
10       ---Tx.2---
11     }
12     ---Tx.3---
13   }
14   Barrier Synchronization
15   for(...){
16     while(...){
17       ---Tx.4---
18     }
19   }
20   Barrier Synchronization
```

図 8　Genome のプログラム構造

```
1  while(...){
2    ---Tx.0---
3    ---Tx.1---
4    ---Tx.2---
5  }
```

図 9　Intruder のプログラム構造

数が STM-default に比して抑えられておらず，競合の発生しにくい Tx であると考えられる．よって実行時間は Tx.2 と同等であるものの，排他実行した場合でも Tx.2 で得られたような速度向上は見られなかったと考えられる．

以上から，Tx が連続的に繰り返し実行されない場合では，実行時間の長い Tx は投機実行した方がよいと考えられる．

2.2.4　排他実行することで性能が向上する Tx の特徴

本稿では，どのような特徴を持つ Tx を排他実行した場合，性能に寄与しうるかについて考察する．

共有変数へのアクセス回数が多い Tx は一般に競合を引き起こす可能性が高く，アボートもされやすいと考えられる．したがって，Kmeans の Tx.0 のような Tx を排他実行することでアボート数を抑制でき，性能が向上すると考えられる．一方，共有変数へのアクセス回数が少ない Tx を排他実行してしまうと，Genome の Tx.0 のように，アボート数を大きく抑制できず，並列度の低下により，高い性能が達成できない．よって，Tx の共有変数へのアクセス回数は，Tx の実行方式の選択にあたり，重要な点の一つである．

次に，Tx が実行されるパターンについて考える．同一 Tx が連続して繰り返し実行される場合，当該 Tx が継続実行される時間帯が長くなるため，当該 Tx 同士で競合が頻発しやすく，Kmeans の Tx.0 のように排他実行することで性能が改善する場合がある．しかしバリア同期等の介在によって，そのような Tx が他 Tx と並行実行される余地が少ない，あるいはない場合は，排他実行により並列性が完全に失われてしまうことで，Genome の Tx.0 のように性能がむしろ悪化する可能性もあり，慎重な見極めが必要である．

最後に，連続的に繰り返し実行されない Tx は，その実行時間が重要となる可能性が高い．実行時間が長く，かつ他 Tx と十分な並列性を持たない Tx は，Intruder の Tx.1 のように，排他実行してしまうとプログラム全体の並列度が大きく低下する．一方で実行時間の短い Tx は，排他実行しても全体の並列度に与える影響は少なく，Intruder の Tx.{0,2} のように排他実行によるアボート抑制が効果的に働く場合がある．

以上から，Tx の実行時間や共有変数へのアクセス回数，Tx 実行のされ方などにより，排他実行が効果的に働くか否かが異なると考えられる．

3.　ロックと TM とを併用した並行性制御手法

本章では，2.1 節で述べたロックと TM の併用に起因する問題を示し，これを解決する新たな並行性制御手法を提案する．

3.1　Locking Transaction

ロックでは，同一のロック変数で保護された区間同士を排他実行させることで，それらの区間を並行実行することを禁じ，共有変数の一貫性を保証している．一方 TM では，Tx 内の共有変数へのアクセスを監視し，複数の Tx を実行中のスレッドで競合が発生した場合には，どちらかの処理を破棄することで一貫性を保証している．

ここで，TM とロックを併用した場合，Tx を実行中のスレッドとロックで保護された区間を実行中のスレッドとが同一の共有変数へアクセスすると，それを競合として検出できず共有変数の一貫性が損なわれてしまう．

この問題に対し，Tx 実行開始時にロックを獲得し，コミットまたはアボート時にロックを解放する並行性制御手法 **Locking Transaction（LTx）**を提案する．LTx は同一 LTx とはロック同様に排他制御されるが，異なる LTx や Tx とは TM 同様，並行実行される．また，LTx は TM の競合検出の機能を備えるため，共有変数の一貫性制御の問題を解決する．

3.2　評価

前節で提案した LTx を，Intel 社の RTM 上に実装し評価を行った．本章では，評価結果を示し，考察する．

3.2.1　評価環境

評価環境は表 1 に示したものと同様である．調査対象のプログラムは STAMP の Kmeans-{low, high} および Genome を用いた．スレッド数は 1, 2, 4, 8, 16, 32 スレッドでそれぞれ実行した．Forward progress は，2.1 節で示

Vol.2018-ARC-230 No.12
Vol.2018-SLDM-183 No.12
Vol.2018-EMB-47 No.12
2018/3/7

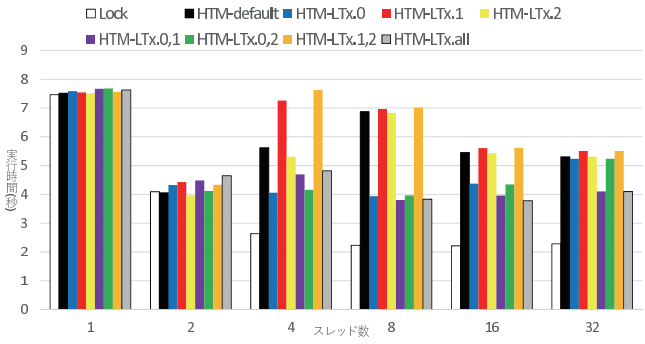

図 10　Kmeans-low

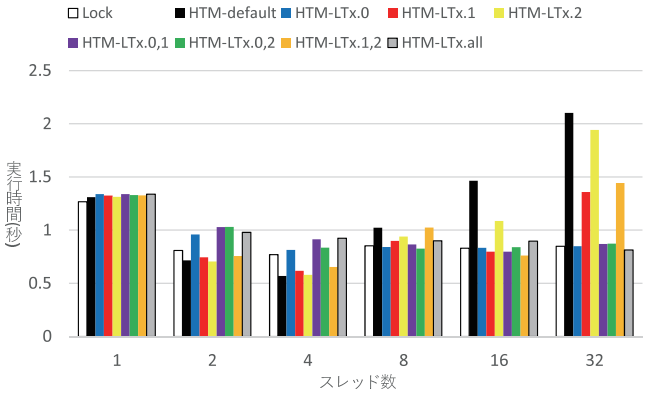

図 11　Kmeans-high

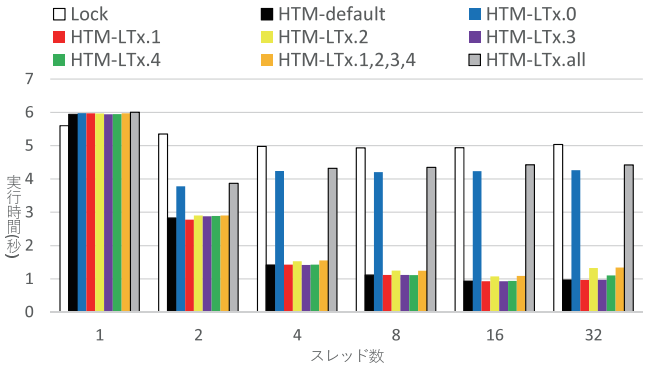

図 12　Genome

したものと同様の方法で保証した．なお，プログラムはそれぞれ 50 回ずつ実行し，その平均を評価結果として示す．

3.2.2　評価結果と考察

各ベンチマークプログラム中の Tx の内の一つまたは複数を LTx にそれぞれ定義し直し，HTM で実行した結果を**図 10** から**図 12** に示す．凡例は HTM-default と Lock は 2.2 節の定義と同様であり，HTM-LTx.n は ID が n の Tx を LTx に定義し直し実行した場合，HTM-LTx.all は全ての Tx を LTx として実行した場合の結果である．

図 10，**図 11** より，Kmeans-{low, high} の HTM-LTx.0 は 2 章の調査で確認できた程の性能は得られていない．これは，LTx の TM 制御オーバヘッドに起因するものであると考えられる。図 10 において，HTM-LTx.all の性能は，HTM-default より優れている場合が多い．これは，同一 LTx 同士の競合がなくなったことで，アボート数が大きく抑制されたためと考えられる．また，HTM-default はスレッド数が変化すると性能が大きく変動している一方，HTM-LTx.all は変動が小さいだけでなく，スレッド数の増加にともない，ほとんどの場合で性能が向上している．図 11 にも似た傾向が見受けられ，スレッド数が少ないと HTM-default の方が性能が優れているが，スレッド数が増えるにつれて HTM-LTx.all の方が高い性能を示している．これらは，スレッド数が増えるに連れて，同時に並行実行される Tx 数の増加により，競合が発生しやすくなり HTM-default は性能が低下したが，LTx では並行実行できる LTx 数に限りがあるため，HTM-LTx.all は性能が低下しなかったためと考えられる．

図 12 に示す Genome の結果からは 2 章で示した調査結果との差異はほとんど確認できない．HTM-default は HTM-LTx.all より高い性能を示しているが，図 6 で示したように，Genome は競合が発生しにくいプログラムであるため，LTx を用いても競合の発生回数はほとんど変化せず，並列度の低下による影響だけが顕在化したためと考えられる．また，HTM-LTx.all が Lock よりも高い性能を示していることから，異なる LTx 同士が TM 同様に並行実行されることが性能に寄与していることがうかがえる。

以上をまとめると，LTx は TM の Tx とロックとの間の性能を持つと考えられ，TM もしくはロックに不向きなプログラムであっても Kmeans-low や Genome のように LTx を用いることである程度の性能を期待できる．また，TM ではスレッド数を増加させることで競合が発生しやすくなり，性能が低下する傾向があるが，Kmeans-{low, high} の結果から分かるように LTx はスレッド数が増加しても性能低下は起こりにくい．また，全てのプログラムの結果より，Tx と LTx を適切に使い分けてプログラムを記述できれば，高い性能が期待できることも確認できる．しかし，Genome に見られるように，すべての Tx を単純に LTx に書き換えた場合，全て Tx のまま実行した場合より性能が劣ることもある．

4.　コード生成手法の設計方針の検討

2.2.4 項で，排他実行することで性能が向上する Tx の特徴について考察した．しかしこれらの特徴を意識し，Tx と LTx のどちらで並行性制御を行う方が性能が向上するかを，定義する区間ごとにプログラマが判断することは困難である．そこで，Tx を用いて記述されたプログラムを解析し，各 Tx に対して Tx と LTx のうちの最適な制御を自動適用するコード生成手法について検討する．

まず，対象の Tx が連続して繰り返し実行される Tx であるかを判別する．連続して繰り返し実行される Tx である場合，その繰り返し実行の前後で同期が行われているか否かを調べる．同期が行われていない場合，この Tx を LTx

としても，他 Tx との並行実行によりある程度の並行性は保たれると考えられる．そこで，当該 Tx 内での共有変数へのアクセス回数がある閾値 n_1 より多いかを調べ，n_1 より多かった場合，競合を頻発させる Tx である可能性が高いとみなし，当該 Tx を LTx として定義しなおす．一方，n_1 より小さかった場合，そのまま Tx として実行する．

一方，同期が行われている場合，当該 Tx の実行時間と，当該 Tx が連続実行される回数との積から計算できる，その Tx が連続して実行される時間帯の長さを計算する．この長さがある閾値 t より長かった場合，当該 Tx を排他実行すると並列度が大きく失われると考え，そのまま Tx として実行する．一方その長さが t より短い場合は，Tx 内での共有変数へのアクセス回数がある閾値 n_2 より多いかを調べ，n_2 より多かった場合，先述した理由から，LTx として定義し直し．n_2 より小さかった場合は Tx のまま定義を変更しない．なお，n_1，n_2，t は環境や TM の実装によって変化すると考えられるため，それらに合った適切な値を設定する必要がある．

最後に，連続的に繰返し実行されない Tx の場合，その実行時間が，同一のループ内で定義されている全ての Tx の実行時間の和より短いなら，この Tx を排他実行しても全体の並列度が完全に損なわれることはないと判断し LTx として定義し直し，長いなら Tx のまま定義を変更しない．

5. 関連研究

Tx 内の一部分又は全てを排他実行することで競合を抑制するという観点から行われた研究として，以下のものが発表されている．Yoo ら [8] は HTM に Adaptive Transactional Scheduling と呼ばれる方式を適用し，競合回数が閾値を超えた場合並列度を低下させることで性能向上を実現している．Blake ら [9] は複数の Tx 内でアクセスされるアドレスの局所性を similarity と定義し，この値が閾値を超えた場合これらの Tx を排他実行することで性能向上を実現している．橋本ら [10][11] は，競合の発生しやすい処理を含む，Tx の一部を排他実行することで性能向上を実現している．

以上の研究は全て，HTM を対象として行われたものであり，専用ハードウェアを追加することで複雑なスケジューリングを実現している．よって HTM 実装の改善に示唆を与えるものであるが，現存の HTM や STM にそのままこれら研究で示されている手法を用いることはできない．そこで本稿では，専用ハードウェアを必要とせず，TM 一般に性能向上を実現しうる手法について検討した．

6. おわりに

本稿では，排他実行することで性能が向上する Tx の特徴について調査した．この結果から，共有変数にアクセスする回数の多い Tx や，特定のパターンで実行される Tx は，当該 Tx 同士を排他実行することによって性能が向上する場合があることが分かった．また，TM とロックを併用することで，共有変数の一貫性が損なわれることを指摘し，これを解決する方法として LTx を提案した．そして，Tx を用いて記述されたプログラムを解析し，プログラム中の各 Tx に対して，Tx と LTx のうちの最適な制御を自動適用するコード生成手法について検討した．

今後は更に多くの環境とプログラムを用いて，より詳細な調査と LTx の評価を行っていくとともに，本稿で示した設計方針に基づくコード生成手法を実装し，評価していきたい．また，Tx と LTx を動的に切り替える手法についても検討する予定である．

謝辞 本研究の一部は，JSPS 科研費 JP17H01711，JP17H01764，および JP17K19971 の助成による．

参考文献

[1] Herlihy, M. et al.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. 20th Int'l Symp. on Computer Architecture (ISCA'93)*, pp. 289–300 (1993).

[2] Minh, C. C. et al.: STAMP: Stanford Transactional Applications for Multi-Processing, *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08)* (2008).

[3] Intel Corporation: *Intel Architecture Instruction Set Extensions Programming Reference, Chapter 8: Transactional Synchronization Extensions*. (2012).

[4] Intel Corporation: *Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 1. Chapter 15: Programming with Intel® Transactional Synchronization Extensions* (2015).

[5] Dice, D., Shalev, O. and Shavit, N.: Transactional Locking II, *Proc. 20th Int'l Conf. on Distributed Computing (DISC'06)*, Springer-Verlag, pp. 194–208 (2006).

[6] Goel, B., Titos-Gil, R., Negi, A., McKee, S. A. and Stenstrom, P.: Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell, *2014 IEEE 28th Int'l Parallel and Distributed Processing Symp.*, pp. 615–624 (2014).

[7] Yoo, R. M., Hughes, C. J., Lai, K. and Rajwar, R.: Performance evaluation of Intel Transactional Synchronization Extensions for high-performance computing, *2013 SC - Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–11 (2013).

[8] Yoo, R. M. et al.: Adaptive Transaction Scheduling for Transactional Memory Systems, *Proc. 20th Annual Symp. on Parallelism in Algorithms and Architectures (SPAA'08)*, pp. 169–178 (2008).

[9] Blake, G. et al.: Bloom Filter Guided Transaction Scheduling, *Proc. 17th Int'l Conf. on High-Performance Computer Architecture (HPCA-17 2011)*, pp. 75–86 (2011).

[10] 橋本高志良，堀場匠一朗，江藤正通，津邑公暁，松尾啓志：Read-after-Read アクセスの制御によるハードウェアトランザクショナルメモリの高速化，情報処理学会論文誌コンピューティングシステム (ACS44)， Vol. 6, No. 4, pp. 58–71 (2013).

[11] Mashita, K., Hirota, A. and Tsumura, T.: Exclusive Control for Compound Operations on Hardware Transactional Memory, *Proc. 2nd IEEE Nordic Circuits and Systems Conf. (NorCAS2016)* (2016).