

メモリアクセス解析に基づく トランザクショナルメモリの ポリシー動的切り替え手法

小林 龍之介¹ 二間瀬 悠希¹ 塩谷 亮太² 五島 正裕³ 津邑 公暁¹

概要：

マルチコア環境では一般に、ロックを用いて共有変数へのアクセスを調停する。しかし、ロックにはデッドロックの発生や並列度の低下などの問題があるため、ロックを代替・補完する並行性制御機構として、トランザクショナルメモリ (TM) が提案されている。この機構をハードウェア上に実装したハードウェアトランザクショナルメモリ (HTM) では、トランザクション (Tx) 同士を投機的に並列実行することで、ロックに比べ並列度が向上する。HTM におけるバージョン管理、および競合検出のための機構には、それぞれに対し Eager/Lazy と呼ばれるポリシーが存在し、それらの組み合わせにより、HTM には3つのポリシーが存在する。これら3つのポリシーはそれぞれ Tx 実行に違いがあり、これが実行時間に与える影響を調査した結果、プログラム毎に最適なポリシーが異なることを確認した。本論文では、トランザクションの持つ特徴と最適なポリシーとの関係を解析して得た指標に基づき、プログラム毎に最適なポリシーに動的に切り替える手法を提案する。評価の結果、使用した全てのプログラムにおいて、より性能の高いポリシーを採用した場合と同程度の性能を達成できることを確認した。

1. はじめに

マルチコア環境では、複数のプロセッサ・コア間で単一アドレス空間を共有する共有メモリ型の並列プログラミングモデルが広く利用されている。このプログラミング方式では、共有変数に対するアクセスを調停する必要がある。それには、これまで一般的にロックが用いられてきた。しかし、ロックを用いた場合、ロック操作のオーバヘッドによる速度性能の低下や、デッドロックの発生などの問題が生じる可能性がある。そのため、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。

そこで、ロックを代替・補完す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。TM は、データベースの更新・検索操作に用いられるトランザクション (Transaction: Tx) の概念をメモリアクセスに適用したものであり、TM を使用する場合は、従来ロックで保護していたクリティカ

ルセクションを含む一連の命令列を、Tx として定義する。そして、共有変数に対するアクセス競合が発生しない限り Tx 同士を投機的に並列実行することで、ロックを用いる場合よりも高い並列性を実現することができる。

なお、TM では Tx の実行が投機的であるため、共有変数の値を更新する際は、更新前と更新後の値を両方保持しておく必要がある (バージョン管理)。また、Tx を実行するスレッド間において、同一アドレスに対するアクセスが競合に相当するか否かを検査する必要がある (競合検出)。ハードウェアトランザクショナルメモリ (Hardware Transactional Memory: HTM) [2], [3], [4] では、これらのための機構をハードウェアで実現することで、Tx 操作によるオーバヘッドを抑制している。

さて、先述の HTM における投機実行のための機構 (競合検出とバージョン管理) には、それぞれに対し Eager/Lazy と呼ばれるポリシーが存在する。そして、それらの4通りの組み合わせの内、現在は3通りの組み合わせがポリシーとして採用されている [5]。これら3つのポリシーはそれぞれ Tx 実行に違いがあり、これがプログラムの実行時間に影響を与えると考えられる。そこで、本論文ではまず、さまざまなプログラムをそれぞれ3つのポリシーで実行し、実行時間を調査する。さらに、各プログラムにおける

¹ 名古屋工業大学
Nagoya Institute of Technology
² 東京大学
The University of Tokyo
³ 国立情報学研究所
National Institute of Informatics

Txの持つ特徴と最適なポリシーとの関係を解析することにより、最適なポリシーを選択するための指標を探る。そして、その指標に基づきプログラム毎に最適なポリシーに動的に切り替える手法を提案し、HTMの性能向上を図る。

2. トランザクショナルメモリ

2.1 トランザクショナルメモリの概要

共有メモリ型並列プログラミングにおいて、TMでは、アクセス競合が発生しない限りTx同士を投機的に並列実行するため、Txを粗粒度に定義したとしても並列度が損なわれることが少なく、記述性に優れる。さらにプログラムは、ロックを使用する際に考慮する必要があるデッドロックを意識すること無くTxを定義できることから、並列プログラムを容易に設計することができる。このTMにおけるTxは以下の2つの性質を満たす必要がある。

Serializability (直列化可能性) 並列実行されたTxの実行結果は、当該Txを逐次実行した場合と同一であり、全てのスレッドから、ある同一の順序で実行されたように観測される。

Atomicity (不可分性) Txはその操作が完全に実行されるか、もしくは全く実行されないかのいずれかでなければならず、各Tx内における処理結果は、Txの完了と同時に観測される。

以上の性質を保証するために、TMではTx内で行われるメモリアクセスを監視する。そして、複数Txからの同一アドレスへのアクセスにより、Txの性質が満たされなくなる場合に、この状態を競合 (Conflict) として検出する。この操作を競合検出 (Conflict Detection) という。競合が検出された場合、いずれかのTxが途中までの処理結果を破棄する。この操作をアボート (Abort) という。これに対し、Tx処理を最後まで完了した場合、Tx内で行った値の更新を確定する。この操作をコミット (Commit) という。なお、Tx実行中はコミットされるか否かが未定であるため、値の更新時には、更新前と更新後の値を両方保持しておく必要がある。そこで、TMでは一方の値を共有メモリに、もう一方の値をそのアドレスと共に別領域に保持する。この操作をバージョン管理 (Version Management) という。また、Txをアボートしたスレッドは、Tx開始前の共有メモリおよびレジスタの状態を復元し、再競合を防ぐためのバックオフ (Backoff) と呼ばれる時間だけ待機した後、Txを再実行する。

2.2 競合検出

2.1節で述べた競合検出は検査のタイミングによって以下の2つに大別される。

Eager Conflict Detection (EagerCD) Tx内でメモリアクセスが発生する度に、そのアクセスが競合に相当するか否かを検査する。

Lazy Conflict Detection (LazyCD) Txのコミットを試みる時点で、そのTx内で行われた全てのアクセスが競合に相当するか否かを検査する。

EagerCDでは、競合が発生した際に即座にそれを検出できるのに対し、LazyCDではコミット時まで競合を検出することができない。それにより、LazyCDではアボート時に無駄になるTx処理が増大してしまう可能性がある。しかし、LazyCDではメモリアクセスの度に競合を検出する必要がないため、競合が発生しない場合にはEagerCDよりオーバーヘッドが小さくなる。

2.3 バージョン管理

2.1節で述べたバージョン管理も、どちらの値を共有メモリに残し、どちらの値を別領域に退避するかによって以下の2つに大別される。

Eager Version Management (EagerVM) 更新前の値を別領域にバックアップし、更新後の値を共有メモリに上書きする。

Lazy Version Management (LazyVM) 更新前の値を共有メモリに残し、更新後の値を別領域にバッファする。

EagerVMでは、コミットは別領域にバックアップした値を破棄するだけで実現できるため高速だが、アボートはバックアップした値を共有メモリに書き戻す必要があるため低速である。それに対し、LazyVMでは、アボートは別領域にバッファした値を破棄するだけで実現できるため高速だが、コミットはバッファした値を共有メモリに上書きする必要があるため低速である。

3. TMにおける3つのポリシー

競合検出およびバージョン管理における各ポリシーの組み合わせにより、TMでは以下の3つがポリシーとして採用されている。

$E_{CD} + E_{VM}$ (EagerCD + EagerVM)

$E_{CD} + L_{VM}$ (EagerCD + LazyVM)

$L_{CD} + L_{VM}$ (LazyCD + LazyVM)

なお、LazyCDとEagerVMを組み合わせたポリシーについては、実装が困難であるため採用されていない。

本章では、各ポリシーの特徴を文献 [6] に示されている代表的な実装に基づいて説明する。

3.1 $E_{CD} + E_{VM}$ の動作

$E_{CD} + E_{VM}$ の代表的な実装では、競合が発生した場合、アクセスリクエストを送信したスレッドは競合相手のTxがコミット、もしくはアボートするまで実行中のTxを中断する。この操作をストール (Stall) という。この動作を図1を用いて説明する。なお、この図は下向きに時間軸をとっており、Core1, Core2上で2つのスレッドThread1,

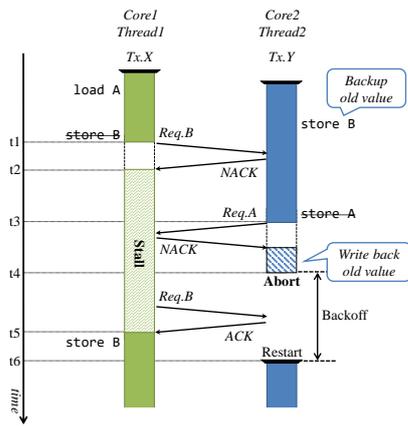


図 1 $E_{CD} + E_{VM}$ における Tx 間の競合解決

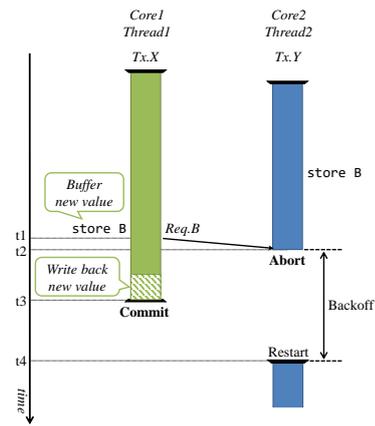


図 2 $E_{CD} + L_{VM}$ における Tx 間の競合解決

Thread2 がそれぞれ $Tx.X$, $Tx.Y$ を実行する例を示している。まず, Thread1 が load A を Thread2 が store B を実行したとする。その後, Thread1 が store B を試み, アドレス B に対するリクエスト $Req.B$ を送信したとする (t1)。しかし, Thread2 は既に当該アドレスに Write アクセス済みであるため競合が検出され, Thread2 は *Nack* を返信し, それを受信した Thread1 は, $Tx.X$ をストールする (t2)。なお, 図中では省略しているが, Thread1 はこの後, アドレス B へのアクセス許可を受けるまで Thread2 に対して定期的リクエストを送信し続ける。実行が進み, Thread2 が store A を試みたとすると (t3), Thread1 は既に当該アドレスに Read アクセス済みであるため競合が検出され, Thread1 は *Nack* を送信する。ここで, Thread2 が $Tx.Y$ をストールすると, Thread1 と Thread2 は互いの Tx の終了を待ち続けるデッドロック状態に陥ってしまう。よってこのような場合, Tx の開始時刻が遅い $Tx.Y$ をアボートすることで, これを回避する (t4)。これにより, Thread1 はアドレス B へのアクセスを許可され, $Tx.X$ をストール状態から復帰させ, 処理を再開する (t5)。一方, $Tx.Y$ をアボートした Thread2 は, バックアップした更新前の値を共有メモリに書き戻し, レジスタ状態も復元した上で, バックオフだけ待機した後, 当該 Tx を再実行する (t6)。

3.2 $E_{CD} + L_{VM}$ の動作

$E_{CD} + L_{VM}$ の代表的な実装では, 競合が発生した場合, アクセスリクエストを受信したスレッドが自身が実行する Tx をアボートする。この動作を図 2 を用いて説明する。まず, Thread2 が store B を実行したとする。次に Thread1 が store B を試み, $Req.B$ を送信したとする (t1)。しかし, Thread2 は既に当該アドレスに Write アクセス済みであるため, 競合が検出される。この際, Thread2 は $Tx.Y$ をアボートする (t2)。実行が進み, Thread1 の $Tx.X$ がコミットを試みたとすると, バッファした更新後の値を共有メモリに上書きし, コミット処理を完了する (t3)。一方, $Tx.Y$ をアボートした Thread2 はバッファした更新後の値

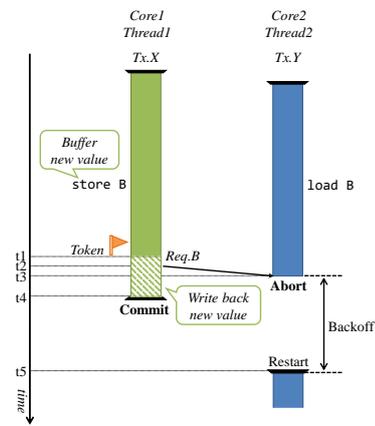


図 3 $L_{CD} + L_{VM}$ における Tx 間の競合解決

を破棄し, レジスタ状態を復元した上で, バックオフだけ待機した後, 当該 Tx を再実行する (t4)。

3.3 $L_{CD} + L_{VM}$ の動作

$L_{CD} + L_{VM}$ の代表的な実装では, 競合が発生した場合, Read アクセスをしており, かつコミット処理中ではない Tx をアボートする。その動作を図 3 を用いて説明する。まず, Thread1 が store B を Thread2 が load B を実行したとする。実行が進み, Thread1 の $Tx.X$ がコミット処理に移ったとすると, 自身がコミット処理を許可されていることを示すトークンの獲得要求をキャッシュディレクトリに対して発行する (t1)。この際, 他スレッドによりトークンが獲得されていなければ, 要求したスレッドがトークンを獲得することができ, このトークンを所持しているスレッドのみが Tx をコミットすることができる。なお, 他スレッドがトークンを獲得していた場合は, トークンの獲得待ちキューに自身のスレッド ID をエンキューし, ストールすることで, 他スレッドの Tx のコミットを待機する。この例では Thread1 がトークンを獲得したとすると, 当該 Tx 内で Write アクセスしたアドレス B に対するリクエスト $Req.B$ を Thread2 に送信する (t2)。しかし, Thread2 は既に当該アドレスに Read アクセス済みであるため競合

表 1 シミュレータ諸元

Processor	SPARC V9
#cores	32 cores
clock	1 GHz
issue width	single
issue order	in-order
non-memory IPC	1
D1 cache	32 KBytes
ways	4 ways
latency	1 cycle
D2 cache	8 MBytes
ways	8 ways
latency	20 cycles
Memory	8 GBytes
latency	450 cycles
Interconnect network latency	14 cycles

表 2 ベンチマークプログラムの入力パラメタ

GEMS microbench	
Btree	priv-alloc-20pct
Contention	config 1
Deque	4096ops 128bkoff
Prioqueue	8192ops
SPLASH2	
Barnes	512
Cholesky	tk14.0
Radiosity	-p 31
Raytrace	teapot
STAMP	
SSCA2	-s13 -i1.0 -u1.0 -l3 -p3
Kmeans	-m40 -n40 -t0.05 -i random-n2048-d16-c16
Vacation	-n2 -q90 -u98 -r16384 -t4096

が検出され、*Thread2* は *Tx.Y* をアボートする (*t3*)。その後、*Thread1* はバッファした更新後の値を共有メモリに上書きし、コミット処理を完了する (*t4*)。一方、*Tx.Y* をアボートした *Thread2* はバッファした更新後の値を破棄し、レジスタ状態を復元した上で、バックオフだけ待機した後、当該 *Tx* を再実行する (*t5*)。

3.4 各ポリシーにおける実行サイクル数の調査

3.1 節~3.3 節で述べた 3 つのポリシーにおける *Tx* 実行の違いは、プログラムの実行時間に影響を与えられられる。そこで、さまざまなプログラムをそれぞれ 3 つのポリシーで実行し、実行時間を比較することで、その影響を調査した。調査には、HTM の研究で広く用いられている LogTM [4] を使用し、シミュレーションを行った。なお、LogTM は $E_{CD} + E_{VM}$ を採用している HTM であるが、それを拡張することで $E_{CD} + L_{VM}$ 、 $L_{CD} + L_{VM}$ における *Tx* 実行を可能にした。

3.4.1 調査環境

調査には Simics [7] 3.0.31 と GEMS [8] 2.1.1 の組合せを用いた。Simics は機能シミュレーションを行うフルシステムシミュレータであり、GEMS はメモリシステムの詳細なタイミングシミュレーションを担う。プロセッサ構成は 32 コアの SPARC V9 とし、OS は Solaris 10 とした。表 1 に詳細なシミュレーション環境を示す。調査対象のプログラムには GEMS microbench, SPLASH-2 [9], および STAMP [10] から計 11 個を使用し、それぞれ 16 スレッドで実行した。表 2 に各プログラムの入力パラメタを示す。

3.4.2 調査結果および考察

調査結果を図 4 に示す。図 4 には、各プログラムの調査結果をそれぞれ 3 本のバーで示しており、各バーは $E_{CD} + E_{VM}$ 、 $E_{CD} + L_{VM}$ 、 $L_{CD} + L_{VM}$ で実行した場合の実行サイクル数を表している。なお、各バーは 10 回の実行における平均を表しており、 $E_{CD} + E_{VM}$ の実行サイク

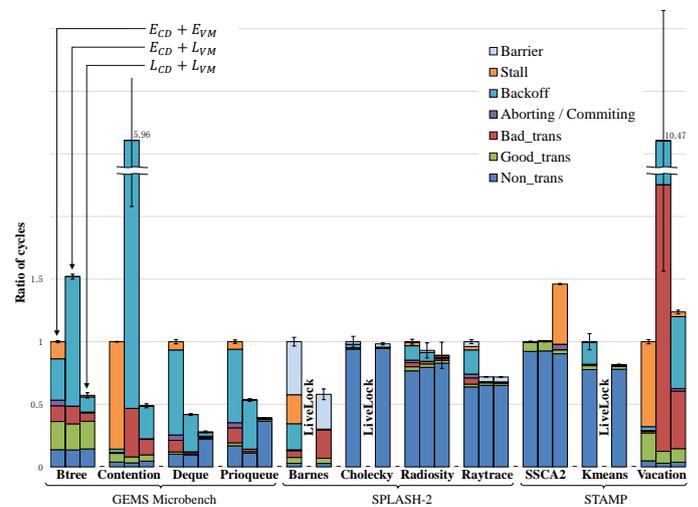


図 4 各プログラムにおける実行サイクル数比

ル数を 1 として正規化している。また、図中のエラーバーは 95% の信頼区間を示しており、凡例は以下のような実行サイクル数の内訳を示している。

Barrier バリア同期に要したサイクル数

Stall ストールに要したサイクル数

Backoff バックオフに要したサイクル数

Aborting/Committing アボート処理またはコミット処理に要したサイクル数

Bad_trans アボートされた *Tx* のサイクル数

Good_trans コミットされた *Tx* のサイクル数

Non_trans *Tx* 外のサイクル数

調査の結果、Barnes, Cholesky, Kmeans において $E_{CD} + L_{VM}$ では LiveLock と呼ばれる現象が発生し、プログラムの実行が終了しなかった。これは、*Tx* 同士が互いに同一アドレスへのリクエストを送信し合うことで、繰り返し互いの *Tx* をアボートさせるという現象である。Barnes, Cholesky, Kmeans 以外の全てのプログラムにおいて $E_{CD} + L_{VM}$ では、そのいずれにおいても実行サイクル数は $E_{CD} + E_{VM}$ または $L_{CD} + L_{VM}$ よりも多く、

$E_{CD} + E_{VM}$ に対して平均 2.70 倍, $L_{CD} + L_{VM}$ に対して平均 3.62 倍であった。よって, $E_{CD} + L_{VM}$ は評価した全てのプログラムに適していないと判断することができる。

また, SSCA2 および Vacation では, $E_{CD} + E_{VM}$ における実行サイクル数が $L_{CD} + L_{VM}$ より平均で 25.4%少ないが, それ以外の全てのプログラムでは, 平均で 36.4%多い。よって, SSCA2 および Vacation は $E_{CD} + E_{VM}$ が適しており, それ以外のプログラムは $L_{CD} + L_{VM}$ が適していると考えられる。

以上のことから, 3つのポリシーにおける Tx 実行の違いはプログラムの実行時間に大きな影響を与え, プログラム毎に最適なポリシーが異なると考えられる。

4. 関連研究

競合検出方式やバージョン管理方式に関する研究として, Tomic ら [5] は EazyHTM (Eager-Lazy HTM) と呼ばれる HTM を提案している。EazyHTM は, 競合検出をメモリアクセスの度に, 競合解決をコミット時に行うことで, 競合の検出と解決のタイミングを分離している。なお, バージョン管理方式は LazyVM を採用している。EazyHTM では, 競合時に相手のコア ID を, 競合相手を記憶するためのテーブルに記憶することで, それ以降そのコアとリクエストを送受信する際に, 本来経由する必要があるキャッシュディレクトリを介さずにコア同士で直接通信を行うことが可能であり, 高速なコア間通信を実現している。また, コミット時には自身のテーブルに存在するコア ID にアポート要求を送信することで, 競合を解決している。本論文では, ポリシーを動的に切り替えることによる性能向上を目標としているが, EazyHTM では, $E_{CD} + L_{VM}$ と $L_{CD} + L_{VM}$ を静的に組み合わせ, 独自の競合解決方式を用いることで HTM の性能向上を実現している。EazyHTM の性能について, Tomic らは $L_{CD} + L_{VM}$ の代表的な実装である TCC [3] との比較をしているが, $E_{CD} + E_{VM}$ との比較がなされていない。そのため, $E_{CD} + E_{VM}$ が適しているプログラムでは, $E_{CD} + E_{VM}$ と EazyHTM における競合解決方式のどちらが適しているかは明らかでない。

また, Lupon ら [11] は Dynamically Adaptable HTM (DynTM) と呼ばれる HTM を提案している。DynTM は, Tx 毎に $E_{CD} + E_{VM}$ と $L_{CD} + L_{VM}$ を動的に切り替え可能なコヒーレンスプロトコルを実装している HTM である。DynTM におけるポリシー切り替えの指標は, Tx のリトライ回数 (ある Tx がコミットするまでに再実行した回数) のみである。しかし, この回数はバックオフアルゴリズムによって大きく変化しうるため, 切り替えの指標としては適切ではないと考えられる。そこで本論文では, 切り替えの指標としてより多くの情報を用いることで, より正確なポリシーの動的切り替えを目指す。

```

1 for (i = 0; i < numOperation; i++){
2     switch (action) {
3         case MAKE_RESERVATION: {
4             BEGIN_TRANSACTION( 0 );
5             long t = types[n];
6             switch (t) {
7                 case RESERVATION_CAR:
8                     /* レンタカーの予約価格の決定 */
9                     break;
10                case RESERVATION_FLIGHT:
11                    /* 飛行機便の予約価格の決定 */
12                    break;
13                case RESERVATION_ROOM:
14                    /* ホテルの予約価格の決定 */
15                    break;
16                default: assert(0);
17            }
18            if (isFound) {
19                /* 顧客情報の登録 */
20            }
21            if (t == RESERVATION_CAR && isFound) {
22                /* レンタカーの予約 */
23            }
24            if (t == RESERVATION_FLIGHT && isFound) {
25                /* 飛行機便の予約 */
26            }
27            if (t == RESERVATION_ROOM && isFound) {
28                /* ホテルの予約 */
29            }
30            COMMIT_TRANSACTION( 0 );
31        }
32        ... /* その他の case */
33    }
34 }
    
```

図 5 Vacation 内の Tx

5. Tx の特徴と最適なポリシーとの関係の解析

3.4 節で述べたように, 3つのポリシーにおける Tx 実行の違いは, プログラムの実行時間に大きな影響を与える。本章では, その原因を探るため, 最適なポリシーが $E_{CD} + E_{VM}$ であった Vacation および SSCA2 を用いて, Tx の持つ特徴と最適なポリシーとの関係を解析する。

5.1 Vacation の解析

Vacation は旅行予約システムの振る舞いを模したプログラムであり, 顧客情報やホテルなどの予約情報を管理する共有データベースに対してアクセスを行う部分が Tx として定義されている。

ここで, Vacation 内に含まれる 3種類の Tx のうち, 総実行回数の約 98%を占める Tx とその前後のコードを図 5 に示す。2行目と 5行目にはそれぞれ switch 文が存在し, それぞれには case 節が 3つずつ含まれるが, どの case 節が実行されるかはランダムに決定される。次に, 具体的な Tx 内の動作について説明する。まず, 4行目で Tx が開始されると, 6行目の switch 文でランダムな値に基づいてい

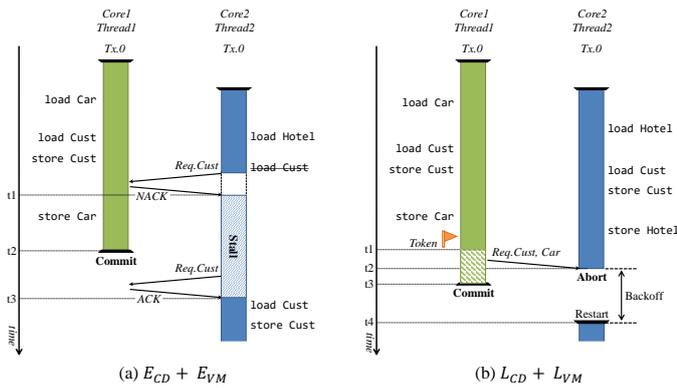


図 6 Vacation における $E_{CD} + E_{VM}/L_{CD} + L_{VM}$ の実行例

ずれかの case 節に入り、レンタカー、飛行機便、ホテルのうち1つの予約価格を決定する。ここでは、データベース内のデータに対して Read アクセスが行われる。次に、18行目の if 文で予約価格の読み出しに成功したか否かを判断し、成功した場合、顧客情報データベースに顧客情報を登録する。ここでは、データベース内のデータに対して Read アクセスおよび Write アクセスが行われる。最後に、3つのうち予約価格を読み出したデータベースに対して、予約の確定操作を行う。ここでは、21~29行目の if 文のうち1つが実行され、データベース内のデータに対して Write アクセスが行われる。

以上の特徴を考慮した上で、この Tx が並行に実行される際、 $E_{CD} + E_{VM}$ と $L_{CD} + L_{VM}$ でどのような違いがあるかを図 6 に示す。なお、各データベースは赤黒木を用いて実装されており、Read アクセス時にはルートノードから各ノードを順に探索するため、複数のアドレスに対してアクセスを行う。また、Write アクセス時には赤黒木全体のバランスを保つためにノードの移動が行われるため、同様に複数のアドレスに対してアクセスを行う。それらのアドレス数は赤黒木の大きさに依存するため、ここではアクセスする複数のアドレスをそれぞれ、Car (レンタカー)、Hotel (ホテル)、Cust (顧客情報) のように1つにまとめて表記する。また、この例では Thread1 がレンタカーの予約を、Thread2 がホテルの予約を試みる場合を示す。

まず、図 6(a) に $E_{CD} + E_{VM}$ における実行例を示す。Thread1 が Tx.0 を開始し、load Car, load Cust, store Cust を実行する。また、Thread2 も同様に Tx.0 を開始し、load Hotel を実行する。ここで、Thread2 が load Cust を試みると、Thread1 の Tx は既に顧客情報データベースに対して Write アクセス済であるため、競合が検出され、Thread2 は Tx をストールする (t1)。その後実行が進み、Thread1 の Tx がコミット処理を行う (t2)。これにより、Thread2 の Tx は顧客情報データベースにアクセスできるようになるため、実行を再開する (t3)。

次に、図 6(b) に $L_{CD} + L_{VM}$ における実行例を示す。まず、Thread1、Thread2 がそれぞれ Tx.0 を開始し、各データ

```

1 int start = chunk * thrID;
2 int stop = start + chunk;
3
4 for(i = start; i < stop; i++){
5     for(j = Graph->outVertexIdx[i]; j < Graph->
6         outVertexIdx[i] + Graph->outDegree[i]; j++){
7         v = Graph->outVertexList[j];
8         BEGIN_TRANSACTION( 2 );
9         int degree = Graph->inDegree[v];
10        Graph->inDegree[v] = degree + 1;
11        EdgeList[v * MAX_CLUSTER_SIZE + degree] = i;
12        COMMIT_TRANSACTION( 2 );
13    }

```

図 7 SSCA2 内の Tx

ベース内のデータに対してアクセスを行う。次に、Thread1 の Tx がコミットを試み、トークンを獲得する (t1)。その際、自身が Tx 内で Write アクセスしたデータベース内のアドレス Car, Cust に対するリクエストを Thread2 に送信する。すると、Thread2 は既に顧客情報データベースに対して Read アクセス済みであるため、競合が検出され、Thread2 は Tx をアボートする (t2)。その後、Thread1 はコミット処理を完了し (t3)、Thread2 はバックオフだけ待機した後、Tx を再実行する (t4)。

以上の Tx 実行の違いを考慮し、 $E_{CD} + E_{VM}$ と $L_{CD} + L_{VM}$ を比較する。まず、 $L_{CD} + L_{VM}$ では、あるスレッドが実行する Tx がコミット間近であったとしても他スレッドからのリクエストにより Tx がアボートされてしまう場合がある。つまり、自身がコミット時に送信するリクエストにより、他スレッドの Tx をアボートさせやすいという特徴を持つ。それに対し、 $E_{CD} + E_{VM}$ では競合が発生したとしても自身の Tx をストールすることにより、それまでに行った Tx 処理を進行させた状態で競合相手の Tx の終了を待機することができるため、高並列な Tx 実行が可能である。そして、競合するまでに行った Tx 処理が多いほど $L_{CD} + L_{VM}$ よりも並列度が高くなると考えられる。

5.2 SSCA2 の解析

SSCA2 (Scalable Synthetic Compact Applications 2) [12] は効率的なグラフデータ構造を構築するプログラムであり、グラフ内の頂点に対する操作が Tx として定義されている。

ここで、SSCA2 に含まれる 3 種類の Tx のうち、総実行回数の 99% 以上を占める Tx とその前後のコードを図 7 に示す。1, 2 行目では、chunk と呼ばれる処理単位を用いて、自身のスレッドが担当する処理範囲を決定する。4 行目に示す for 文では、その処理範囲を順にたどるようにイテレータ変数 i の値が増加する。5 行目に示す for 文では、イテレータ変数 i を用いて、共有リストから操作する頂点

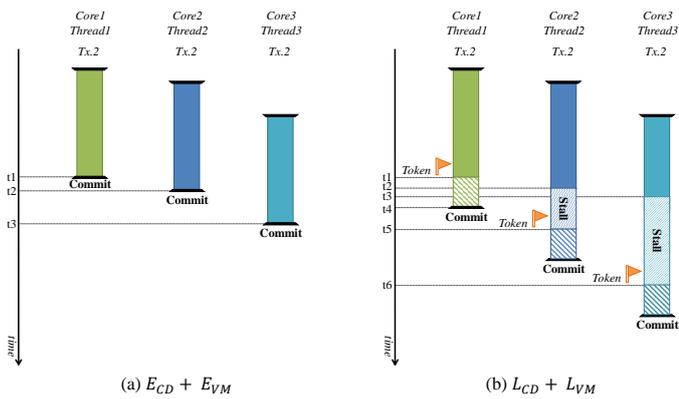


図 8 SSCA2における $E_{CD} + E_{VM}/L_{CD} + L_{VM}$ の実行例

を選択し、その頂点の出次数の大きさだけイテレータ変数 j の値を増加させる。6 行目では、イテレータ変数 j を用いて頂点リストから操作する頂点 v を得る。7 行目で T_x が開始されると、8 行目で先ほど選択した頂点 v の入次数を読み出し、9 行目でその値を 1 増加させる。その後、10 行目で頂点 v に辺を追加する。

以上の特徴を考慮した上で、この T_x が並行に実行される際、 $E_{CD} + E_{VM}$ と $L_{CD} + L_{VM}$ でどのような違いがあるかを図 8 に示す。なお、6 行目における変数 v のアドレスは for 文の各イテレータ変数によって常に変化し、 T_x 同士が同一アドレスにアクセスする可能性が低いため、8、9 行目の Read アクセスおよび Write アクセスではほとんど競合が発生しない。また、 T_x 内で行われるメモリアクセス回数は他のプログラムに比べて少ない（平均 4.99 回）ため、全スレッドの T_x が頻繁にコミットを試み、そのタイミングが重なりやすい。そこで、図 8 では競合が発生せず、コミットのタイミングが他スレッドの T_x と近い場合を考える。

まず、図 8(a) に $E_{CD} + E_{VM}$ における実行例を示す。 $Thread1$ 、 $Thread2$ 、 $Thread3$ がそれぞれ $T_x.2$ を開始するが、この例では競合が発生しないため、 $t1 \sim t3$ において各スレッドにおける T_x がコミットすることができる。

次に、図 8(b) に $L_{CD} + L_{VM}$ における実行例を示す。 $Thread1$ 、 $Thread2$ 、 $Thread3$ がそれぞれ $T_x.2$ を開始し、 $Thread1$ の T_x が最初にコミットを試みたとする。このとき、 $Thread1$ はトークンを獲得し、コミット処理に移る ($t1$)。次に、 $Thread2$ の T_x がコミットを試みたとすると、この時点では $Thread1$ がトークンを獲得しているため、 $Thread2$ はトークン獲得待ちキューに自身のスレッド ID をエンキューし、 T_x をストールする ($t2$)。ここでさらに、 $Thread3$ の T_x がコミットを試みたとすると、同様に T_x をストールする ($t3$)。実行が進み、 $Thread1$ がコミット処理を完了したとする ($t4$)。それにより、トークン獲得待ちキューの先頭にあるスレッド ID に該当する $Thread2$ はトークンを獲得し、コミット処理に移ることができる ($t5$)。その後、 $Thread2$ がコミット処理を完了すると、 $Thread3$

表 3 各プログラムにおける指標に関する調査結果

benchmark	Prog	#AbortTx	Qlen	最適ポリシー
Btree	1.51 %	4.85	0.48	$L_{CD} + L_{VM}$
Contention	1.56 %	9.33	0.12	$L_{CD} + L_{VM}$
Deque	4.25 %	1.31	0.00	$L_{CD} + L_{VM}$
Prioqueue	1.97 %	0.65	0.14	$L_{CD} + L_{VM}$
Barnes	7.45 %	0.91	1.61	$L_{CD} + L_{VM}$
Cholecky	1.37 %	2.36	0.28	$L_{CD} + L_{VM}$
Radiosity	5.65 %	1.03	0.26	$L_{CD} + L_{VM}$
Raytrace	1.16 %	0.91	0.01	$L_{CD} + L_{VM}$
SSCA2	0.00 %	0.88	11.52	$E_{CD} + E_{VM}$
Kmeans	1.25 %	0.64	0.38	$L_{CD} + L_{VM}$
Vacation	46.31 %	9.36	0.00	$E_{CD} + E_{VM}$

はトークンを獲得し、コミット処理に移る ($t6$)。

以上の T_x 実行の違いを考慮し、 $E_{CD} + E_{VM}$ と $L_{CD} + L_{VM}$ を比較する。まず、 $E_{CD} + E_{VM}$ では T_x がストール/アボートすることがなく、高並列な T_x 実行が可能である。それに対し、 $L_{CD} + L_{VM}$ では T_x 内でアクセス競合が発生しないにもかかわらず、 T_x がコミットを試みるタイミングが重なりやすいことにより、トークンの獲得待ちが連鎖的に発生し、並列度が低下してしまう。

5.3 ポリシー切り替えの指標

5.1 節～5.2 節の解析結果から、以下の 3 つがポリシーを切り替える際の指標となる可能性があり、各値が他のプログラムと比べて大きいプログラムは $E_{CD} + E_{VM}$ における T_x 実行の利点が活きるため、 $E_{CD} + E_{VM}$ が適していると考えられる。

- $E_{CD} + E_{VM}$ における、初めて競合するまでに行った T_x 処理の、 T_x 処理全体に対する進行度。（初めて競合するまでに行ったメモリアクセス回数を、その T_x 内の総メモリアクセス回数で割った値。以下、*Prog* とする）
- $L_{CD} + L_{VM}$ における、コミット時にアボートさせた他スレッドの T_x 数。（以下、 $\#AbortTx$ とする）
- $L_{CD} + L_{VM}$ における、コミットを試みた際のトークン獲得待ちキューに存在するスレッド ID 数。（以下、*Qlen* とする）

これらが適切な指標となり得るかを調査するために、各プログラムにおける 3 つの値をそれぞれ計測した。なお、調査環境は 3.4.1 項に示したものと同様である。調査結果を表 3 に示す。なお、各値はコミットされた全 T_x について平均をとったものであり、最右列に図 4 で確認した最適ポリシーを示している。

表 3 より、 $\#AbortTx$ は最適なポリシーが $E_{CD} + E_{VM}$ ではないプログラムにおいても高い数値を示す場合があることから、*Prog* および *Qlen* の 2 つを用いることで、最適なポリシーの選択が可能であると考えられる。

6. ポリシー動的切り替え手法

6.1 動作概要

提案手法では、並列に動作する複数のスレッドのうち、1つのスレッド（以下、Master_Coreで動作するMaster_Threadとする）に *Prog* および *Qlen* の値の計測を任せ、その値が予め定義した閾値を超えた場合に、ポリシーを切り替えさせる。

まず、ポリシーを切り替えさせる際の動作について述べる。異なるポリシーで動作するTxが混在する環境では競合検出が困難であるため、ポリシーの切り替えはTx単位ではなくプログラム単位で行う必要がある。そこで本手法では、ポリシー切り替えの条件が満たされた際、現在実行中のTxが全て完了するのを、バリア同期を用いて待機した後、全コアにポリシー切り替えを指示することでこれを実現する。具体的には、Master_Threadはポリシーの切り替えを判断した際に、他スレッドに対して待機状態に移ることを依頼するメッセージ（以下、*Wait* メッセージとする）を送信する。そして、それを受信した各スレッドは、Txをコミットまたはアポートすると待機状態に移る。その際、各スレッドはMaster_Threadに対して、待機状態に移ったことを報告するメッセージ（以下、*Waiting* メッセージとする）を送信する。他スレッド全てからそのメッセージを受信し、変更前のポリシーで動作するTxが存在しないことをMaster_Threadが確認すると、全コアにポリシー切り替えを指示する。その後、他スレッドに対して実行を再開させるメッセージ（以下、*Restart* メッセージとする）を送信し、それを受信した各スレッドは自身の処理を再開する。なお、本手法では、多くのプログラムで $L_{CD} + L_{VM}$ が適していることを考慮し、プログラム開始時は $L_{CD} + L_{VM}$ で動作させ、指標値が条件を満たした際に $E_{CD} + E_{VM}$ へと切り替えさせる。

次に、各値の計測方法の概要について述べる。*Prog* を計測するためには、Tx内でどのメモリアクセスが最初に競合したかを知る必要がある。そのためには、メモリアクセスの度に競合を検出する必要があり、これはEagerCDの動作に該当する。しかし本手法では、 $L_{CD} + L_{VM}$ で動作しながらその値を計測する必要があるため、 $L_{CD} + L_{VM}$ を拡張し、メモリアクセスの度に他スレッドに対して競合検出を依頼するメッセージ（以下、*Check* メッセージとする）を送信する。そして、これを受信した各スレッドは競合の発生を検査し、競合した場合は競合したことを表すメッセージ（以下、*Conflict* メッセージとする）を、競合しなかった場合は *Ack* を返信する。なお、 $L_{CD} + L_{VM}$ で動作している間は、競合解決を行うのはコミット時である。一方、*Qlen* を計測するためには、コミット時にトークン獲得のために待機しているスレッド数を知る必要がある。

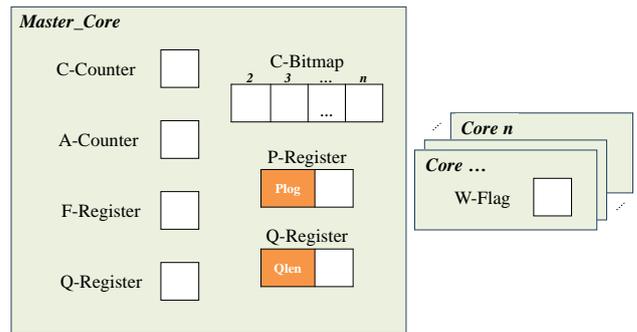


図9 拡張したハードウェア構成

これを実現するために、各スレッドはトークンの獲得要求をした際にキャッシュディレクトリに対してキューサイズを問い合わせるメッセージ（以下、*Get_Qlen* メッセージとする）を送信する。

6.2 ハードウェア構成

提案手法を実現するために、既存のHTMを拡張し図9に示すように7つのハードウェアをMaster_Coreに追加する。また、ポリシー切り替え時のバリア同期のため、Master_Core以外のコアに、*Wait* メッセージを受信したことを表すフラグを追加する。追加したハードウェアの詳細を以下に示す。なお、図中の n は総コア数を示している。

コミット回数カウンタ (C-Counter) : Master_ThreadでコミットしたTx数を記憶する。

メモリアクセスカウンタ (A-Counter) : Tx内で発生したメモリアクセス回数を記憶する。

初競合メモリアクセスレジスタ (F-Register) : Tx実行中に初めて競合した際のメモリアクセスカウンタの値を記憶する。

トークン獲得待ちスレッド数レジスタ (Q-Register) : コミット時にトークンの獲得要求をした際に、キャッシュディレクトリに対して *Get_Qlen* メッセージ送信し、トークン獲得待ちスレッド数を記憶する。

コアビットマップ (C-Bitmap) : ポリシー切り替えの際のバリア同期時に、Master_Threadが *Waiting* メッセージを受信した際、そのメッセージの送信元スレッドのコアIDに該当するビットをセットする。他の全スレッドから *Waiting* メッセージを受信し、コアビットマップ内のビットが全てセットされれば、Master_Threadは他の全スレッドが待機状態に移ったことを確認することができる。

Prog 記憶レジスタ (P-Register) : コミット回数カウンタを用いて算出した *Prog* の値を記憶する。なお、算出する値はプログラム開始からの平均である。

Qlen 記憶レジスタ (Q-Register) : コミット回数カウンタを用いて算出した *Qlen* の値を記憶する。上記同

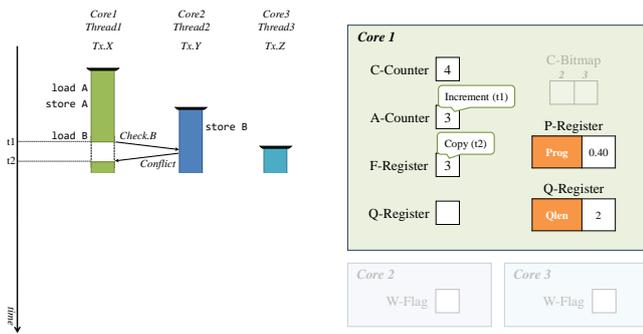


図 10 初競合時のメモリアクセスカウンタの値を記憶する動作

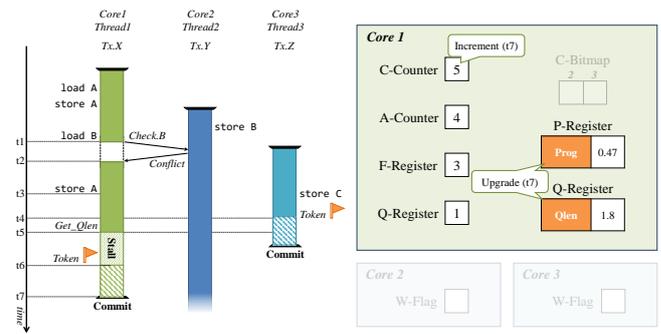


図 12 各記憶レジスタの平均値を更新する動作

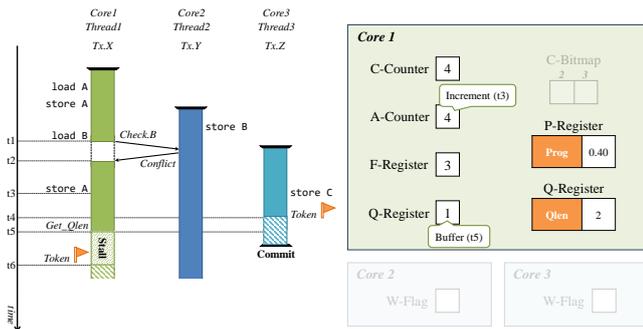


図 11 トークン獲得待ちキューに存在する待機スレッド数を記憶する動作

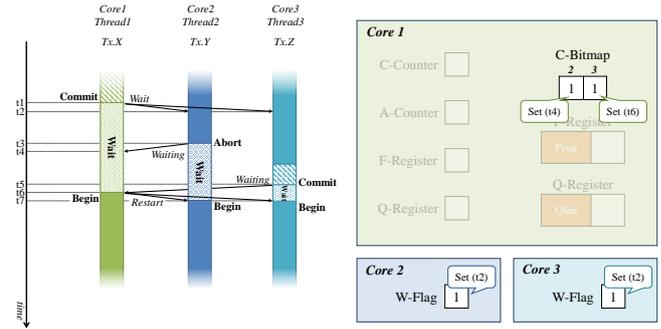


図 13 ポリシー切り替え時の動作

様、算出する値はプログラム開始からの平均である。

待機フラグ (W-Flag) : Master_Thread から Wait メッセージを受信した際にセットされる。コミット時またはアボート時に自身の待機フラグがセットされていた場合、Master_Thread から Restart メッセージを受信するまで待機状態に移る。

6.3 動作モデル

本節では、Master_Core を Core1 に、Master_Thread を Thread1 に割り当てた場合の、提案手法の動作モデルを示す。

6.3.1 各記憶レジスタの平均値を更新する動作

各記憶レジスタの平均値を更新する動作について図 10～図 12 を用いて説明する。なお、既にコミット回数カウンタには 4、各レジスタには Prog: 0.40, Qlen: 2 という値がそれぞれ記憶されていることとする。

まず、図 10 に示すように、Thread1, Thread2, Thread3 がそれぞれ Tx.X, Tx.Y, Tx.Z を開始し、Thread1 が load A, store A を、Thread2 が store B を実行したとする。その際、Thread1 はメモリアクセス毎にメモリアクセスカウンタの値をインクリメントする。次に、Thread1 が load B を試みたすると、メモリアクセスカウンタの値を 3 にインクリメントし、他スレッドに Check.B メッセージを送信する (t1)。このメッセージを受信した Thread2 は、既に当該アドレスに Write アクセス済みであるため、Thread1 に Conflict メッセージを返信する。これを受信した Thread1

は、初競合メモリアクセスレジスタに現在のメモリアクセスカウンタの値である 3 を記憶する (t2)。

実行が進み、図 11 に示すように、Thread1 が store A を実行したとする (t3)。その際、Thread1 はメモリアクセスカウンタの値を 4 にインクリメントする。また、Thread3 が store C を実行した後トークンを獲得し、コミット処理に移ったとする (t4)。その後、Thread1 が実行する Tx.X がコミット時にトークンの獲得を試みると、この時点では Thread3 がトークンを獲得しているため、Thread1 は Tx.X をストールする (t5)。その際、Thread1 はキャッシュディレクトリに対して Get_Qlen メッセージを送信することで、トークン獲得待ちキューのサイズ 1 を取得し、トークン獲得待ちスレッド数レジスタに記憶する。Thread3 がコミット処理を完了すると、Thread1 はトークンを獲得後にコミット処理を開始する (t6)。

さらに実行が進み、図 12 に示すように、Thread1 がコミット処理を完了したとする (t7)。その際、コミット回数カウンタの値を 5 にインクリメントする。ここで、Thread1 は Tx 実行中に記憶した各値に基づき、各記憶レジスタの平均値を更新する。

6.3.2 ポリシー切り替え時の動作

次に、ポリシー切り替え時の動作を図 13 を用いて説明する。なお、図 13 は、Thread2, Thread3 がそれぞれ Tx を実行中に Thread1 に計測した指標値が条件を満たした例を示している。まず Thread1 は、Thread2, Thread3 に Wait メッセージを送信する (t1)。これを受信した Thread2, Thread3 は各自の待機フラグをセットする (t2)。

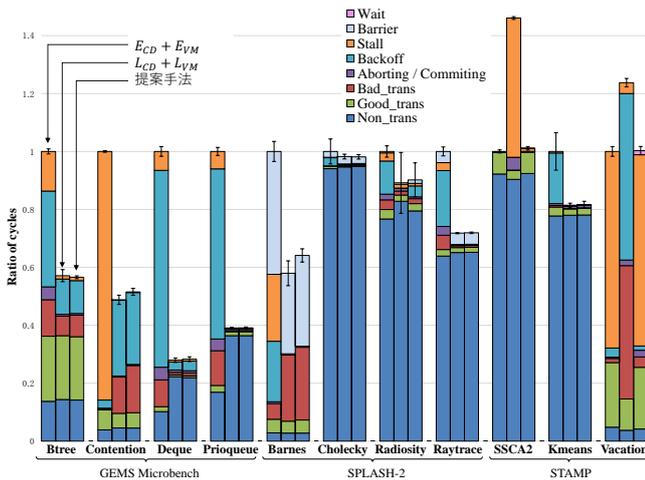


図 14 各プログラムにおける実行サイクル数比

実行が進み、*Thread2* が実行する *Tx.Y* がアボートした際、待機フラグがセットされているため、*Thread2* は待機状態に移る (t3)。その際 *Thread2* は、*Thread1* に *Waiting* メッセージを送信する。これを受信した *Thread1* は、コアビットマップ内の *Core2* に該当するビットをセットする (t4)。

さらに実行が進み、*Thread3* の実行する *Tx.Z* がコミットすると、待機状態に移る (t5)。また、*Thread3* は、*Thread1* に *Waiting* メッセージを送信し、*Thread1* は、コアビットマップ内の *Core3* に該当するビットをセットする (t6)。ここで、コアビットマップにおける、プログラムを実行中の他コアに該当する全てのビットがセットされていることを *Thread1* が確認すると、全コアにポリシー切り替えを指示する。そして、*Thread1* は、*Thread2*、*Thread3* に *Restart* メッセージを送信し、自身の処理を再開する。また、それを受信した *Thread2*、*Thread3* も処理を再開する (t7)。

6.4 評価

提案手法を LogTM に実装し、シミュレーションにより評価した。なお、評価環境は 3.4.1 項に示したものと同様である。

6.4.1 評価結果および考察

各プログラムにおける実行サイクル数比を図 14 に示す。図 14 では、図 4 で示した $E_{CD} + E_{VM}$ 、 $L_{CD} + L_{VM}$ に加え、提案手法を適用したモデルの実行サイクル数を加えた計 3 本のバーを示している。また、凡例には、

Wait 提案手法で追加した待機に要したサイクル数を追加した。なお、図 4 同様、それぞれのバーは 10 回の実行における平均を表しており、 $E_{CD} + E_{VM}$ の実行サイクル数を 1 として正規化している。

まず、SSCA2 および Vacation では、最適なポリシーである $E_{CD} + E_{VM}$ に動的に切り替えたことで、提案手法における実行サイクル数が $E_{CD} + E_{VM}$ における実行サイクル数と同等であることが確認できる。また、ポリシー切り

替え時の待機に要したサイクル数は少ないことから、これがプログラムの実行に与える影響は少ないと考えられる。

次に、SSCA2 および Vacation 以外の全てのプログラムでは、提案手法において終始 $L_{CD} + L_{VM}$ で実行されたが、メモリアクセス毎に競合検出をしたことで、それに起因するサイクル数の増加がみられるプログラムが存在する。しかし、提案手法における実行サイクル数は $L_{CD} + L_{VM}$ における実行サイクル数と同等であり、増加したサイクル数はわずかであることがわかる。

以上のことから、提案手法を適用することで、最適なポリシーでプログラムを実行可能であり、提案手法に起因するオーバーヘッドはわずかであることが確認できる。

6.4.2 ハードウェアコスト

本節では、提案手法を実現するために追加したハードウェアの実装コストについて述べる。なお、各コストは、今回評価に用いた全てのプログラムを 16 スレッドで実行する場合を想定し、算出した。

初めに、各指標値を計測するために必要なハードウェアのコストについて述べる。まず、コミット回数カウンタには、プログラム実行中の総コミット数を記憶可能なビット幅が必要となる。そこで、今回評価に用いた全てのプログラムについて調査したところ、最大数は約 100000 であったため、必要なビット幅は 17bit となる。次に、メモリアクセスカウンタには、各 Tx におけるメモリアクセス回数の総数を格納する必要がある。初競合メモリアクセスレジスタはメモリアクセスカウンタの最大値と同一の値を記憶する可能性がある。そこで、今回評価に用いた全てのプログラムについて調査したところ、最大数は 94451 であった。よって、それぞれに必要なビット幅は 17bit となる。トークン獲得待ちスレッド数レジスタには、16 スレッドを並列実行する場合、最大で 15 スレッドが獲得待ちとなることから、この値を記憶可能な 4bit が必要となる。最後に、Prog 記憶レジスタおよび Qlen 記憶レジスタには、それぞれ倍精度浮動小数点型の値を保存可能な領域が必要であり、それぞれに 64bit となる。

次に、ポリシー切り替え時のバリア同期に必要なハードウェアのコストについて述べる。コアビットマップには、16 スレッドを並列実行する場合 Master_Core 以外のコア数に対応する 15bit が必要となる。また、待機フラグには Master_Core 以外のコアに 1bit ずつ、つまり 15bit 必要となる。

以上より、提案手法を実現するために必要となるハードウェアコストは、16 スレッドを実行可能な 16 コア構成のプロセッサにおいて 26.625Byte (213bit) であり、ごく小容量のハードウェアで提案手法を実現することができる。

7. おわりに

本論文では、HTM における 3 つのポリシーの Tx 実行

の違いがプログラムの実行時間に影響を与えることと、プログラム毎に最適なポリシーが異なることを確認した。また、Txの持つ特徴と最適なポリシーとの関係を解析して得た指標に基づき、プログラム毎に最適なポリシーに動的に切り替える手法を提案した。

提案手法の有効性を確認するために、既存のHTMを拡張しシミュレーションにより評価した。その結果、全てのプログラムにおいて、より性能の高いポリシーを採用した場合と同程度の性能を達成できることを確認した。また、提案手法を実現するために必要な追加ハードウェアのコストを概算したところ、26.625Byteであり、ごく小容量のハードウェアでこの手法を実現できることを示した。

今後の課題として、異なるポリシーで動作するTx同士を並列実行可能なコヒーレンスプロトコルを実装し、Tx毎に最適なポリシーに切り替えることで、Txの種類が複数存在するプログラムにおいてもスループットの向上を図ることが挙げられる。

謝辞 本研究の一部は、JSPS 科研費 JP17H01711, JP17H01764, JP17K19971 の助成を受けたものである。

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] Knight, T.: An Architecture for Mostly Functional Languages, *Proc. ACM Conference on LISP and Functional Programming (LFP'86)*, pp. 105–112 (1986).
- [3] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C. and Olukotun, K.: Transactional Memory Coherence and Consistency, *Proc. 31st Annual Int'l Symp. Computer Architecture (ISCA'04)*, pp. 102–113 (2004).
- [4] Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D. and Wood, D. A.: LogTM: Log-based Transactional Memory, *Proc. 12th Int'l Symp. on High-Performance Computer Architecture (HPCA'06)*, pp. 254–265 (2006).
- [5] Tomic, S., Perfumo, C., Kulkarni, C., Arnejach, A., Cristal, A., Unsal, O., Harris, T. and Valero, M.: Eazyhtm, Eager-lazy Hardware Transactional Memory, *Proc. 42nd Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-42)*, pp. 145–155 (2009).
- [6] Bobba, J., Moore, K. E., Volos, H., Yen, L., Hill, M. D., Swift, M. M. and Wood, D. A.: Performance Pathologies in Hardware Transactional Memory, *Proc. 34th Annual Int'l Symp. on Computer Architecture (ISCA'07)*, pp. 81–91 (2007).
- [7] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hällberg, G., Högberg, J., Larsson, F., Moestedt, A. and Werner, B.: Simics: A Full System Simulation Platform, *Computer*, Vol. 35, No. 2, pp. 50–58 (2002).
- [8] Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D. and Wood, D. A.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset, *ACM SIGARCH Computer Architecture News*, Vol. 33, No. 4, pp. 92–99 (2005).
- [9] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd Annual Int'l. Symp. on Computer Architecture (ISCA'95)*, pp. 24–36 (1995).
- [10] Minh, C. C., Chung, J., Kozyrakis, C. and Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing, *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08)* (2008).
- [11] Lupon, M., Magklis, G. and González, A.: A Dynamically Adaptable Hardware Transactional Memory, *Proc. 43rd Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-43)*, pp. 27–38 (2010).
- [12] Bader, D. A. and Madduri, K.: Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors, *International Conference on High-Performance Computing*, Springer, pp. 465–476 (2005).