Vol.2022-ARC-248 No.46
Vol.2022-SLDM-198 No.46
Vol.2022-EMB-59 No.46
2022/3/11

分散トランザクショナルメモリのための ソフトウェアキャッシュの設計と実装

山下 淳[1]　二本松 秀樹[1]　山本 和諒[1]　浅井 優太[1]　塩谷 亮太[2]　五島 正裕[3]　津邑 公暁[1]

概要：現代の科学技術分野の進歩を支える高性能な大規模並列計算基盤は分散メモリ型と呼ばれるハードウェアの形態を取る．この形態を前提としたシステムでは，高性能なプログラムを実現するために，長い期間を費やして手動でチューニングする必要があり，生産性が問題となっている．このような中，生産性と性能を両立しうる機構としてトランザクショナルメモリ（TM）への関心が高まっている．TM はマルチコアプロセッサ向けに多く研究されてきたが，この機構を分散システムにも適用することで，分散メモリ型におけるプログラミングを単純にすることも可能であるという考えから，分散システムを対象とする分散トランザクショナルメモリ（DTM）が提案されているが，実用的な実装は未だ存在していない．我々は生産性と性能を両立する大規模並列計算基盤を実現するため DTM に着目し，生産性の高いプロトタイプを開発したが，性能面に改善の余地がある．そこで本稿では，DTM の性能を高めるためのソフトウェアキャッシュを設計および実装する．提案する DTM システムの有効性を確認するため，マイクロベンチマークおよび STAMP ベンチマークを用いて評価した結果，プロトタイプと比較してマイクロベンチマークでは 1.56 倍，STAMP では 3.64 倍の高速化を確認した．

1. 序論

現代の科学技術の発展には，莫大な計算量を含む精密かつ膨大なシミュレーションが不可欠であり，それを実現するのが京 [1] や富岳 [2] などのスーパーコンピュータを代表とする大規模計算基盤である．このような計算基盤は**分散メモリ型**の構造を取り，今日に至るまでの数十年間その性能を向上させ続けてきた．一方で問題となっているのが，分散メモリ型における開発の生産性である．分散メモリ型の並列計算機では，タスクやデータを計算機へどのようにマッピングするかを細かく指定する必要があり，最大性能を出すためのチューニング作業も容易ではないことから，性能のために生産性が犠牲になっている．この問題に対し，**分散共有メモリ** [3–6] や **Partitioned Global Address Space（PGAS）** [7–13] などが提案・開発されてきたが，いまだ生産性と性能の両立を実現できてはいない．

このような中，生産性と性能を両立しうるプログラミングパラダイムとして，**トランザクショナルメモリ（Transactional Memory: TM）** [14] が注目を集めている．TM は共有メモリに対する同期を抽象化する機構であり，従来同期に使用されてきたロックを代替・補完することが期待されている．TM では従来ロックで保護されていたクリティカルセクションを含む一連の処理を**トランザクション（Transaction: Tx）**として定義する．そしてメモリアクセスが競合しない限り Tx 同士を投機的に並列実行する．TM はデッドロックなどの並列プログラム固有の問題が発生しないことから生産性に優れるうえ，Tx を投機的に並列実行することから，ロックと比較して高い並列度を実現できる場合が多い．

こうした特徴から，TM は主に共有メモリ型アーキテクチャを対象として研究・開発されてきた．しかし，マルチノード向けの研究成果は乏しく，実用的な実装は未だ存在していない．そこで我々は，PGAS モデルに TM による同期機構を導入し，マルチノード向け TM を開発することで，生産性と性能を両立する計算基盤の実現を目指してきた。現在はプロトタイプの開発が完了しており，既存のマルチコアプロセッサ向け TM と同様の高い生産性を実現しているが，性能面に改善の余地がある．そこで本稿では，マルチノード向け TM に適したソフトウェアキャッシュを設計および実装することで，プロトタイプよりも性能を高め，マルチノード向け TM の実用性向上を目指す．

[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

Vol.2022-ARC-248 No.46
Vol.2022-SLDM-198 No.46
Vol.2022-EMB-59 No.46
2022/3/11

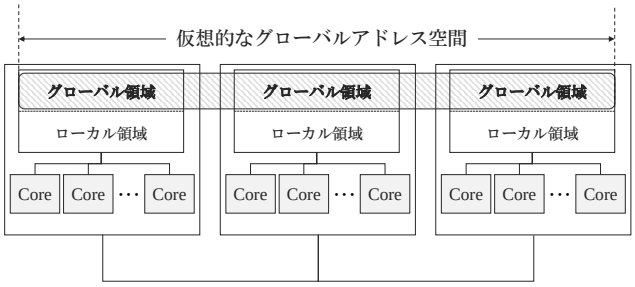


図 1 Partitioned Global Address Space

2. 研究背景

本章では，従来の大規模計算基盤について説明した後，TM の概要について述べる．

2.1 従来の大規模計算基盤

従来の大規模計算基盤は，単一のメモリ空間を共有するマルチコアプロセッサ（ノード）を，高速なネットワークで相互接続した分散メモリ型の構造をとる．別のノードのメモリ上にあるデータにアクセスする際は，あるノードが別のノードを指定してデータを送受信する必要がある．また，すべての計算機を活用するためには，単一ノードでは考慮する必要がなかった，データやタスクの分散が必要であり，データ構造から変更しなければならない．そのため，既存の逐次プログラムを書き直す作業は，非常にコストが高い．さらに，ハードウェアを最大限活用して高性能を実現するためには，生産性を犠牲にした綿密なチューニングが必要である．このように，分散メモリ型の開発環境では，性能のために生産性が犠牲になっている．

この問題に対する様々な研究が行われており，その 1 つが PGAS である．PGAS とは，**図 1** のように各ノードのメモリがグローバル領域とローカル領域に分割されており，全てのプロセッサは他のノード上のグローバル領域に対しても読み書きが可能というモデルである．グローバル領域へのアクセス方法は PGAS 処理系によって異なるが，get/put のようなインタフェースを提供し，プログラマに明記させることが多い．この場合，get はグローバル領域からローカル領域への転送，put はローカル領域からグローバル領域への転送を意味する．

一般的な PGAS においては，キャッシュ機構は導入されていない．そのためキャッシュの機能を組み込みたい場合は，その実装およびキャッシュに対するコヒーレンス制御はプログラマの責任である．チューニングの余地があるため，最大性能は高いが，その分プログラマへの負担が大きい．また，キャッシュの実装などは，アプリケーションのアルゴリズムと直接関係のないものであり，プログラマが本来すべき仕事ではない．このような点から，PGAS においても生産性の問題を十分に解決できているとは言えない．

2.2 Transactional Memory

このような中，性能と生産性を両立しうるプログラミングパラダイムとして，TM が注目を集めている．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にくい．さらにプログラマは，ロックを用い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なく 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．

TM における Tx は以下の性質を満たす必要がある．

直列化可能性（Serializability）：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，全てのスレッドにおいて同一の順序で実行されたように観測される．

不可分性（Atomicity）：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ある．

以上の性質を保証するために，TM は Tx 内で発生するメモリアクセスを監視する．複数の Tx 内で同一アドレスに対するアクセスが確認され，これらのアクセスによって上記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 が途中までの実行結果を破棄する．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 内で行われ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 Tx 内で更新したデータ，もしくは更新前のデータをそのアドレスとともに別領域に退避する．このようなデータの管理を**バージョン管理（Version Management）**という．

上記のような競合検出およびバージョン管理のための機構はハードウェアもしくはソフトウェアによって実装される．ハードウェアによって実装された TM は**ハードウェアトランザクショナルメモリ（Hardware Transactional Memory: HTM）**という．HTM は専用の記憶領域とフラグをプロセッサに搭載することで TM の機能を実現する．記憶領域で記憶できるアクセス数には上限があるが，ソフトウェアによる実装と比較してより高速に実行できる．これに対してソフトウェアによって実装される TM は**ソフトウェアトランザクショナルメモリ（Software Transactional Memory: STM）**という．STM ではメタデータを利用して競合を検出するため，HTM と比較して競合検出に係るオーバヘッドが大きいが，HTM とは異なり容量等のハードウェアによる種々の制約がない．

このような利点から TM は主にマルチコアプロセッサ

Vol.2022-ARC-248 No.46
Vol.2022-SLDM-198 No.46
Vol.2022-EMB-59 No.46
2022/3/11

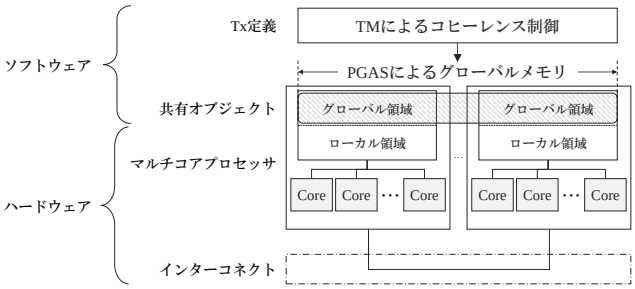


図 2 計算基盤の構成図

向けに広く研究・開発されてきた [15–20]．この TM の機構を分散システムにも適用することで，分散メモリ型の開発環境におけるプログラミングを単純にすることも可能であるという考えから，分散システムをターゲットとする**分散トランザクショナルメモリ（Distributed Transactional Memory: DTM）**の研究もいくつか行われているが [21–24]，実用的な実装は未だ存在していない．

3. 開発中の DTM

我々は生産性と性能を両立する大規模計算基盤を実現するため，DTM に着目し，実用性を備えた DTM を目指して開発を進めてきた．開発中の計算基盤の構成を**図 2** に示す．本計算基盤は二つの層によって構成される．ハードウェア層は，マルチコアプロセッサがインターコネクトで接続されたアーキテクチャで構成される．ソフトウェア層は，PGAS 処理系によるノードをまたがる仮想的なグローバルメモリ空間と，DTM によるコヒーレンス制御で構成される．そして，ユーザはノード間で共有する共有オブジェクトおよび DTM が提供する Tx 定義を利用することで，アプリケーションを記述する．このような方針のもとで開発を進め，逐次プログラムと同等の高い生産性を実現するプロトタイプを開発した．

しかし，性能面に課題が残っており，性能を向上させる方法について検討してきた結果，これまでの設計では二つの点から性能向上が困難であることが次第にわかってきた．1 つ目は，通信回数の多さである．これまでの設計では，グローバルメモリに対するアクセスの度に，**Remote Procedure Call（RPC）**による通信が発生する．そのため，Tx 内でアクセスする共有オブジェクトの数が多いほど，通信回数が増大し，性能が低下しやすい．2 つ目は，競合検出にかかるオーバヘッドの大きさである．これまでの設計では，既存の STM で用いられてきた競合検出を，マルチノードで実行できるように拡張してきた．しかし，2.2 節で述べたように，STM は競合検出にかかるオーバヘッドが大きく，既存の STM をベースとしたプロトタイプもその影響を受けている．そこで，本研究では，プロトタイプの設計を見直すことでこれらの問題点を解決し，DTM の性能向上を目指す．

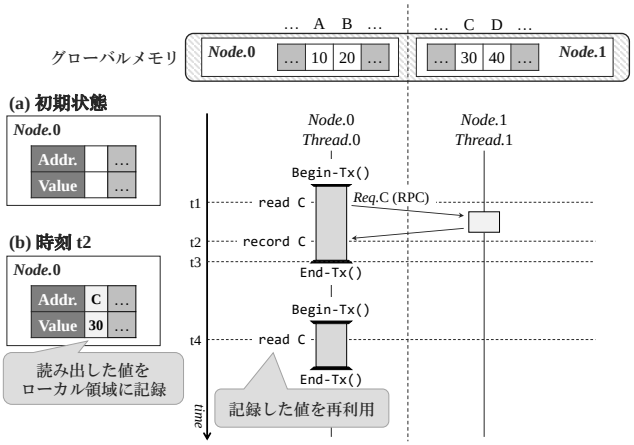


図 3 他のノードから読み出した値を記録して再利用

4. 新たなシステムの設計方針

本章では，DTM の性能を向上させるにあたり，3 章で述べた二つの問題点を解決する方法について述べる．まず 1 つ目の問題点である，RPC による通信回数の多さを解決する方法を検討する．PGAS モデルでは，グローバルメモリ空間の一部が各ノードの物理メモリにマッピングされており，他のノードのメモリにマッピングされているグローバルアドレスにアクセスする度に通信が発生する．

そこで，他のノードにマッピングされているアドレスにアクセスした際に読み出した値を，それぞれのノードのローカル領域で記録しておくこととする．そして，当該アドレスの値が書き換えられていない場合，記録した値を再利用することで，RPC による通信回数を削減することが期待できる．この様子を**図 3** を用いて説明する．図では，*Node*.0 の *Thread*.0 が，Tx 内で *Node*.1 にマッピングされているアドレス C を読み出す様子を表している．図に示すように，*Node*.0 は *Node*.1 からのリプライを受信した際に，アドレス C とその値をローカル領域のテーブルに記録する．そして，以降の読み出しでは，テーブルから値を読み出すことで，RPC を発行せずに処理を進行できる．しかし，記録した値が他のノードで実行される Tx によって書き換えられていたとすると，古い値を使用して Tx 内の処理を進行させてしまう．そのため，記録した値の一貫性を保証する必要がある．

次に 2 つ目の問題点である，競合検出にかかるオーバヘッドの大きさを解決する方法を検討する．なお，説明を簡単にするため，DTM ではなくマルチコアプロセッサ向け STM を想定して説明する．STM ではメタデータを利用して競合を検出しており，多くの STM はメタデータとしてバージョン情報を利用する．具体的には，スレッド間で共有している変数にバージョンを付与し，変数を更新する際にはそのバージョンも更新する．各スレッドは変数を読み出した際にその時点でのバージョンを記録しておき，

Vol.2022-ARC-248 No.46
Vol.2022-SLDM-198 No.46
Vol.2022-EMB-59 No.46
2022/3/11

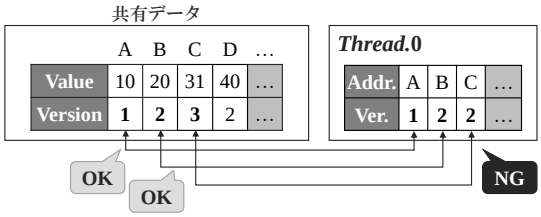


図 4 読み出した全ての変数のバージョンを順に比較

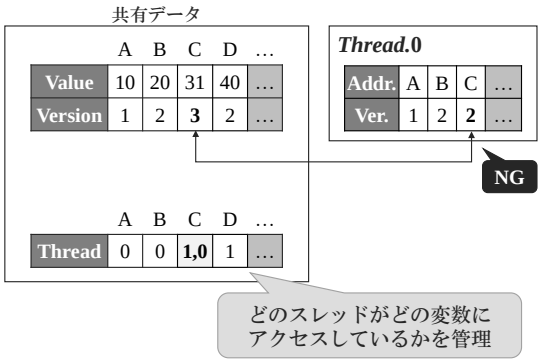


図 5 競合が発生している可能性のある変数のバージョンのみ比較

最新のバージョンと比較する．バージョンが更新されていた場合，他の Tx によって変数が書き換えられていることを検出し，Tx をアボートする．このとき，他のスレッドによってどの変数が更新されているかがわからないため，**図 4** に示すように，Tx 内で読み出した全ての変数のバージョンを順に比較しなければならない．

そこで，どのスレッドがどの変数にアクセスしているかという情報を管理することとする． そして，他のスレッドが実行する Tx によって値が書き換えられている可能性のある変数のみ競合の有無を確認することで，競合検出にかかるオーバヘッドの削減が期待できる．この様子を**図 5** を用いて説明する．図に示すように，*Thread*.0 がバージョンを記録している変数のうち，変数 C のみ他のスレッド（この例では *Thread*.1）によってアクセスされていることがわかるため，変数 C のバージョンのみ比較すればよい．

以上をまとめ，本研究では，他のノードにマッピングされているアドレスから読み出した値を記録するための機構としてキャッシュメモリを，どのスレッドがどのアドレスにアクセスしたかを記録するための機構としてディレクトリをソフトウェアによって導入し，DTM のためのソフトウェアキャッシュを設計および実装することで，DTM の高速化を目指す．

5. ソフトウェアキャッシュの設計

本章では，ソフトウェアキャッシュの設計について論ずる．ソフトウェアキャッシュを設計するにあたり，本稿では，Intel 社および IBM 社の実プロセッサに搭載されている HTM の動作を参考にする．HTM はキャッシュメモリのハードウェア拡張およびコヒーレンスプロトコルを利用

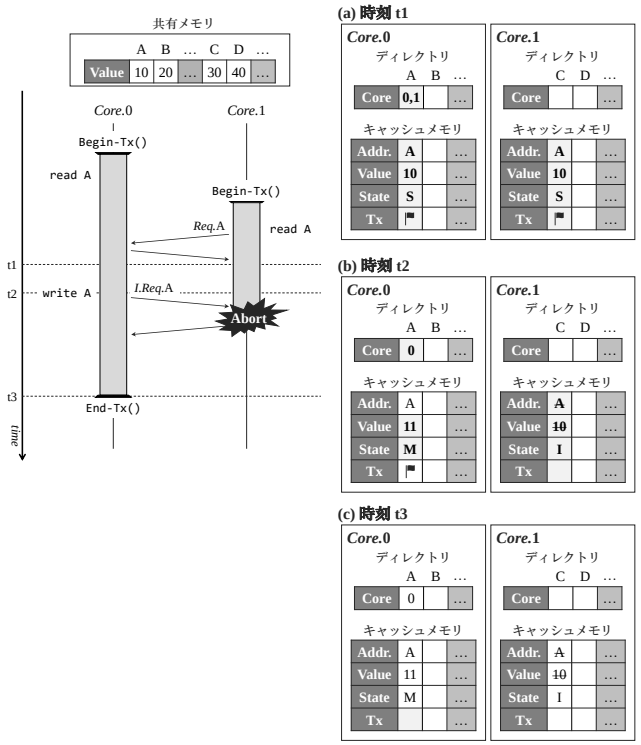


図 6 HTM の動作例

することで TM の機能を実現しているため，この知見を活用することで，DTM に適したソフトウェアキャッシュを設計する．

5.1 キャッシュを利用する HTM の動作

ハードウェアキャッシュを利用する HTM の動作を**図 6** を用いて説明する．なお，コヒーレンスプロトコルは MESI プロトコル [25] とする．HTM では，TM の機能を実現するため，各キャッシュラインにビット拡張が施されている（図中キャッシュメモリの Tx 行）．このビットは，当該キャッシュラインが Tx 内でアクセスされたかどうかを区別するために設けられており，Tx 内でアクセスされた場合にセットされる．本稿では，このビットを **Tx ビット**と呼ぶこととする．この Tx ビットにコヒーレンスプロトコルを組み合わせることで，競合を検出する．

まず，*Core*.0 および *Core*.1 が Tx 内で `read A` を実行したとする（t1）．このとき，各コアのキャッシュラインは Shared 状態かつ Tx ビットがセットされているキャッシュラインとなり（図 6(a)），Tx 内で read アクセスされたキャッシュラインであることがわかる．次に，*Core*.0 が `write A` を実行したとすると（t2），コヒーレンスプロトコルにしたがい，*Core*.0 は *Core*.1 に無効化リクエストを送信する．リクエストを受信した *Core*.1 は，Shared 状態かつ Tx ビットがセットされているキャッシュラインに対する無効化リクエストであることから，Tx 内で読み出したアドレスに対する書き込みが行われたことがわかるため，競合を検出する．*Core*.1 はキャッシュラインを無効化

Vol.2022-ARC-248 No.46
Vol.2022-SLDM-198 No.46
Vol.2022-EMB-59 No.46
2022/3/11

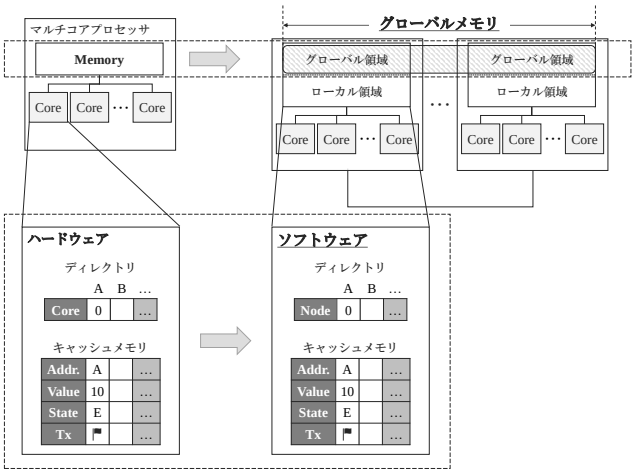
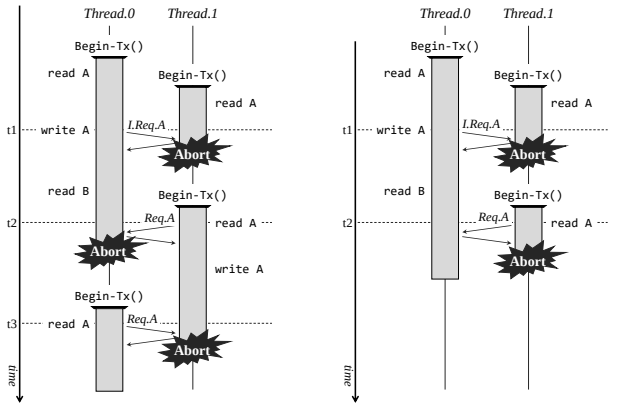


図 7 ソフトウェアによる HTM のエミュレート



(a) 後からアクセスしたTxが処理を継続　(b) 先に書き込んだTxが処理を継続

図 8 競合解決ポリシー

し，全ての Tx ビットをクリアすることで（図 6(b)），Tx をアボートする．その後，*Core*.0 が Tx 内の処理を完了したとすると（t3），全ての Tx ビットをクリアすることで（図 6(c)），Tx をコミットする．このように，HTM ではハードウェアキャッシュを利用することで，共有メモリに対するアクセス回数を削減しつつ，共有メモリの一貫性を保証している．

そこで本稿では，HTM を実現するためのハードウェアキャッシュの動作をソフトウェアでエミュレートすることを考える．**図 7** に示すように，マルチコアプロセッサにおける共有メモリを，本計算基盤におけるノードをまたがるグローバルメモリと捉え，マルチコアプロセッサ内の各コアが持つハードウェアキャッシュメモリおよびディレクトリを，各ノードが持つ物理メモリのローカル領域にそれぞれ実装する．また，コヒーレンスプロトコルの動作を，ノード間の通信を利用してソフトウェアで実装する．このようにすることで，グローバルメモリに対するコヒーレンス制御を提供しつつ，通信回数の削減が期待できる．以降の節では，TM の性能およびコヒーレンス制御に影響を与える，競合が発生した際にどちらの Tx をアボートするか，およびソフトウェアキャッシュのコヒーレンスプロトコルについて検討する．

5.2 競合解決ポリシーの設計

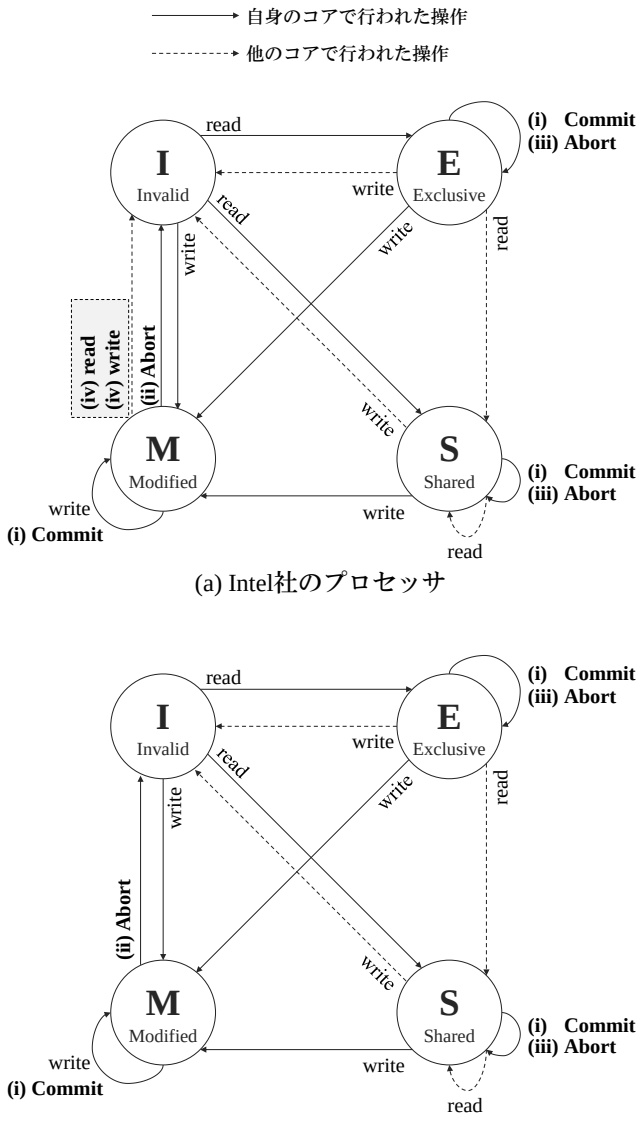
2.2 節で述べたように，TM では競合が発生すると，どちらかの Tx をアボートすることで，プログラムの処理を続行させる．どちらの Tx をアボートするかは競合解決ポリシーと呼ばれる．本節では，本稿で開発する DTM において採用する競合解決ポリシーについて論ずる．ここでは，実プロセッサに搭載されている HTM のうち，**図 8** に示す二種類のポリシーについて検討する．

1 つ目は，後からアクセスした Tx が処理を継続し，アクセスリクエストを受信した Tx がアボートする requester-win と呼ばれるポリシーである（図 8(a)）．このポリシーは，Intel 社のプロセッサで採用されている．アクセスリクエストを受信した Tx がアボートするため，実装が容易である．一方で，アボートした Tx を再実行すると，再び同一のメモリアドレスにアクセスすることから，Tx 同士が互いにアクセスリクエストを送受信しあい，アボートし続けるライブロックと呼ばれる状態に陥る可能性がある．

2 つ目は，あるアドレスに先に書き込んだ Tx が処理を継続し，後から同一アドレスに対して read アクセスまたは write アクセスを試みた Tx がアボートするポリシーである（図 8(b)）．このポリシーは，IBM 社の POWER プロセッサで採用されている．アクセスリクエストを受信した Tx ではなく，アクセスリクエストを送信した Tx をアボートできるようにする必要があるため，requester-win よりも実装が複雑になる．一方で，ライブロックが発生しにくいという利点がある．

以上の動作を踏まえたうえで，DTM に適すると考えられる競合解決ポリシーを検討する．一般に，コア間の通信はレイテンシがナノ秒のオーダーであるのに対し，ノード間の通信はレイテンシがマイクロ秒からミリ秒のオーダーであることから，DTM においてライブロックに陥った場合，そのオーバヘッドは HTM と比較して，極めて大きなものとなることが考えられる．したがって，本稿では，図 8(b) に示すポリシーを採用する．このポリシーを実現するため，それぞれの計算ノードに確保されるディレクトリの各エントリに，ロック変数を用意することとする．そして，Tx 内で読み出しおよび書き込みを行う際は，アクセスしたいアドレスに対応するエントリのロック状態を確認する．ロックが獲得されていない場合，そのアドレスには他の Tx によって書き込みが行われていないことがわかるため，読み出しであれば値を読み出し，書き込みであればロックを獲得することでそれ以降の読み出しおよび書き込みを防ぐ．ロックを獲得した Tx は，コミットまたはアボートの際にロックを解放することで，他の Tx からのアクセスを可能にする．一方で，ロックが獲得されている場

Vol.2022-ARC-248 No.46
Vol.2022-SLDM-198 No.46
Vol.2022-EMB-59 No.46
2022/3/11



(a) Intel社のプロセッサ

(b) IBM社のPOWERプロセッサ

図 9　HTM で Tx を実行している最中の状態遷移図

合，他の Tx によって書き込みが行われていることがわかるため，競合を検出し，Tx をアボートする．

5.3　状態遷移および Write back の設計

4 章で述べたように，キャッシュから古い値を読み出して処理が進行することを避けるため，各ノードが持つソフトウェアキャッシュ間でキャッシュした値の一貫性を保証する必要がある．本節では，これに必要なソフトウェアキャッシュのコヒーレンスプロトコルについて検討する．

5.3.1　HTM におけるコヒーレンスプロトコル

実プロセッサでは MESI プロトコルをベースとしたプロトコル [26,27] を利用しているため，本稿でも MESI プロトコルを想定して設計を進める．このプロトコルを採用しているプロセッサの HTM で Tx を実行する場合，コミットおよびアボートによる新たな状態遷移が追加され，一部の動作が変更される．Intel 社のプロセッサで Tx を実行している最中の状態遷移図を図 9(a) に，IBM 社の POWER プロセッサで Tx を実行している最中の状態遷移図を図 9(b) に示す．これらのプロセッサでは，競合解決ポリシーの違いにより，共通している動作と異なる動作が存在する．まず，共通している動作について説明する．Tx をコミットする場合，Tx 内でアクセスしたキャッシュラインの Tx ビットをクリアするだけであるため，各キャッシュラインの状態を維持する（図 9(i)）．また，Tx をアボートする場合，状態が Modified であるキャッシュラインについては，Tx 内で実行した書き込み結果を破棄する必要があるため，状態を Invalid に遷移させる（図 9(ii)）．一方で，状態が Exclusive あるいは Shared であるキャッシュラインについては，読み出しのみが行われており，当該キャッシュラインを無効化する必要はないため，状態を維持する（図 9(iii)）．

次に，異なる動作について説明する．Intel 社のプロセッサでは，競合解決ポリシーに requester-win を採用しているため，図 9(a)(iv) に示すように，状態が Modified かつ Tx ビットがセットされているキャッシュラインに対して，他のコアが read または write の実行を試みると，Tx をアボートし，当該キャッシュラインの状態を Invalid に遷移させる．一方で，IBM 社の POWER プロセッサでは，あるアドレスに対して先に書き込みを実行した Tx が処理を継続する競合解決ポリシーを採用しているため，図 9(b)(i) および (ii) に示すように，状態が Modified かつ Tx ビットがセットされているキャッシュラインについては，他のコアで行われた操作によって状態が遷移することはなく，自身のコミットまたはアボートでのみ状態が遷移する．

5.3.2　DTM に適したコヒーレンスプロトコル

本稿では，IBM 社の POWER プロセッサと同じ競合解決ポリシーを採用しているため，図 9(b) に示す状態遷移をそのまま採用すればよいかのように思える．しかし，新たに考慮しなければならない Tx の実行パターンが存在する．それは，共有メモリに書き戻していないキャッシュラインに対して，Tx 内で書き込みを実行する場合である．ここまで述べてきたように，Tx がアボートすると，状態が Modified であるキャッシュラインの状態を Invalid に遷移させることで，Tx 内で実行した書き込み結果を破棄する．このとき，当該キャッシュラインが共有メモリに書き戻していないキャッシュラインであったとすると，キャッシュラインの状態を Invalid に遷移させたことで Tx を実行する前に行われていた書き込み結果が失われてしまい，Tx 開始前の状態を復元できなくなる．

IBM 社の POWER プロセッサでは，このような状況が発生し，書き込み結果が失われることを防ぐため，対策が取られている [28]．POWER プロセッサでは，L2 キャッシュで競合検出が行われており，共有メモリに書き戻していないキャッシュラインに対して Tx 内で書き込みを行う場合，L3 キャッシュに書き込み前の値を退避する．このとき，当該 Tx がコミットあるいはアポートするまで，退避

Vol.2022-ARC-248 No.46
Vol.2022-SLDM-198 No.46
Vol.2022-EMB-59 No.46
2022/3/11

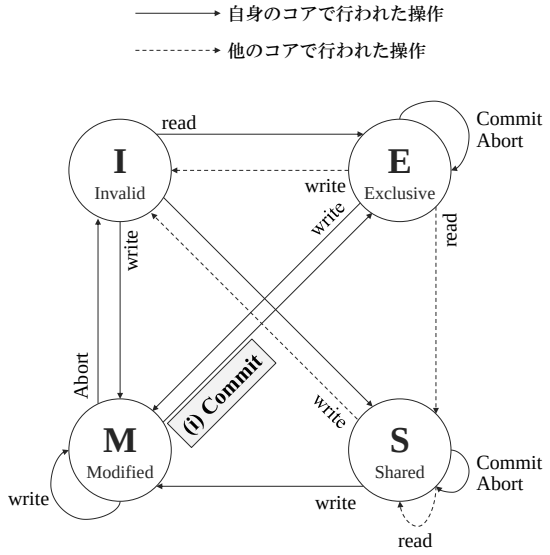


図 10 提案システムで Tx を実行している最中の状態遷移図

した値を他のコアから読み出せないようにしておく．そして，Tx をコミットできた場合は退避した値を破棄し，アボートした場合は L2 キャッシュの当該キャッシュラインを無効化し，以降は L3 キャッシュに退避した値を再利用する．なお，共有メモリに書き戻さず，L3 キャッシュに退避するのは，共有メモリに対するアクセス回数を削減し，高速なキャッシュ上で処理を完結させるためである．

一方で，本稿の DTM の場合，他のノードが持つグローバルメモリに対するアクセスおよび，他のノードが持つソフトウェアキャッシュに対するアクセスは，どちらも同じインターコネクトを介して行われるため，これらのアクセス速度は同一である．そのため，Tx をコミットした際に，グローバルメモリに値を書き戻さずに保持しておく利点がないのみならず，ソフトウェアキャッシュ間の一貫性を保証するための制御が複雑になることで，ノード間の通信回数が増加し，却って性能が悪化してしまう可能性がある．そこで，コヒーレンスプロトコルの一部を改変し，Tx をコミットする際は，Tx 内で書き込んだ値を全てグローバルメモリに書き戻し，キャッシュラインの状態を Modified から Exclusive に遷移させるようにする．このように改変したプロトコルを**図 10** に示す．基本的には POWER プロセッサと同様の状態遷移を行うが，図中の (i) に示すように，Tx をコミットする場合，状態が Modified であるキャッシュラインについては，グローバルメモリに値を書き戻し，状態を Exclusive に遷移させる．このようにすることで，Tx をコミットした際はグローバルメモリ上の値が常に最新となるため，キャッシュとメモリの値が異なることで生じる可能性のある状況が発生しなくなる．本稿では，以上のように状態遷移および Write back を設計し，ソフトウェアキャッシュ間の一貫性を制御する．

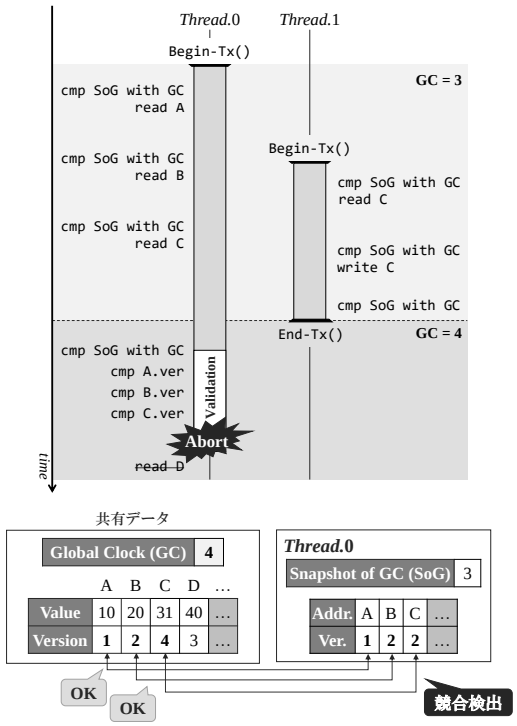


図 11 従来の STM における競合検出（Validation）

6. 競合検出のオーバヘッド削減方法

本章では，ソフトウェアキャッシュに含まれるディレクトリを用いて，競合検出のオーバヘッドを削減する方法について論ずる．

6.1 STM の競合検出

マルチコアプロセッサ向け STM では一般に，**図 11** に示すように，システム全体の論理時刻を表す Global Clock，および変数ごとに付与したバージョン情報を利用して競合を検出する．Global Clock は Tx がコミットするごとにインクリメントされ，Tx をコミットした際に，変数のバージョンを Global Clock の値で更新する．各スレッドは，変数を読み出す際にその時点でのバージョンを記録しておく．また，Tx 開始時の Global Clock を Snapshot として退避しておき，変数にアクセスする前および Tx をコミット前に最新の Global Clock を確認する．Global Clock が更新されていた場合，Tx 内で読み出した変数が書き換えられている可能性があるため，記録したバージョンと最新のバージョンとを比較する．もしバージョンが更新されていた場合，自身が読み出してから他の Tx によって変数が書き換えられたことがわかるため，競合を検出する．このとき，他のスレッドによってどの変数が更新されているかがわからないため，それまでに記録した全ての変数のバージョンを比較する必要がある．このように，バージョンを順に比較して競合を検出する処理のことを，STM では一般に **Validation** と呼ぶ．

Vol.2022-ARC-248 No.46
Vol.2022-SLDM-198 No.46
Vol.2022-EMB-59 No.46
2022/3/11

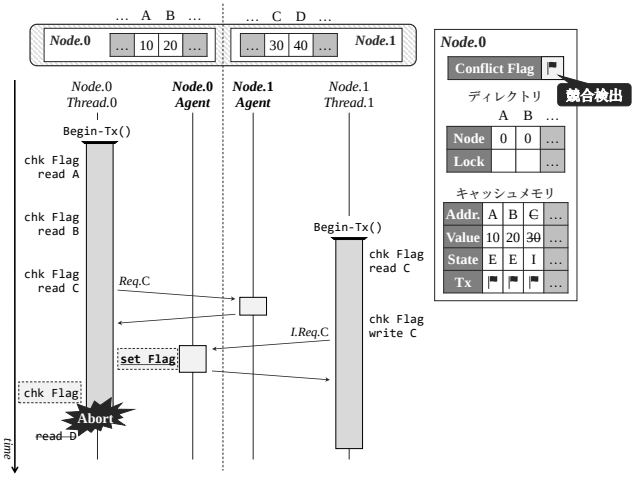


図 12 提案システムにおける競合検出

ここで，Validation に要する計算量を求める．最良のケースは，Tx 実行中を通じて Global Clock が更新されていない場合である．この場合，共有変数に対する各アクセスの際に Global Clock と Snapshot とを比較するだけでよいため，Tx 内で行われるアクセスの回数を n 回とすると，計算量は $O(n)$ となる．一方で，最悪のケースは，Tx 実行中を通じてアクセスの度に Global Clock が更新されており，なおかつ競合が発生していなかった場合である．この場合，アクセスの度に，それまでに記録した全ての変数のバージョンを比較しなければ，競合が発生していないことを保証できない．したがって，Tx 内で行われる n 回目のアクセスでは，$n-1$ 個のバージョンを比較しなければならず，計算量は $O(n^2)$ となってしまう．一般に，Tx を実行している最中に Global Clock が更新される確率は高く，平均の計算量も $O(n^2)$ となることが知られている．DTM においては，論理時刻をノード間で共有することが困難であるため，我々が開発したプロトタイプでは，コミット直前に一度だけ行う方法を採用している．この場合，Validation の回数を削減できるものの，競合が発生していた場合に Tx 全体の実行が破棄されてしまうという問題点がある．

6.2 提案システムの競合検出

提案システムでは，ソフトウェアキャッシュを利用し，Tx 内でアクセスしたキャッシュラインが無効化されたかどうかで競合を検出するため，Validation が不要となる．ただし，HTM とは異なりソフトウェアにより競合を検出するため，Tx を実行するスレッドにキャッシュラインが無効化されたかどうかを通知する必要がある．

そこで，提案システムでは，**図 12** に示すように，他のノードからのアクセスリクエストを受信しソフトウェアキャッシュを制御するスレッド，および Tx を実行するスレッドに競合が発生したか否かを通知するためのフラグを，それぞれ各ノードに設ける．本稿では，これらを ***Agent Thread*** および **Conflict Flag** と呼ぶこととする．*Agent Thread* はアクセスリクエストを受信し，それが Tx ビットがセットされているキャッシュラインに対する無効化のリクエストであった場合，Conflict Flag をセットする．Tx を実行するスレッドは，グローバルメモリに対するアクセスを行う前およびコミット前に Conflict Flag がセットされているかを確認するだけで，競合を検出することができる．そのため，Tx 内で行われるアクセスの回数を n 回とすると，どのようなケースでも計算量は $O(n)$ となる．したがって，提案システムでは常に従来の STM における最良時の計算量で済むため，競合検出にかかるオーバヘッドを削減し，高速化に繋がることが期待できる．

```
1  for (int i = 0; i < NUM_TX; i++) {
2      BEGIN_TX();
3      int idx;
4      int value;
5      for (int j = 0; j < read_only; j++) {
6          if (rand100() < locality) {
7              idx = rand_local();
8          } else {
9              idx = rand_remote();
10         }
11         value = TX_READ(array[idx]);
12     }
13     idx = rand_local();
14     value = TX_READ(array[idx]);
15     value++;
16     TX_WRITE(array[idx], value);
17     END_TX();
18 }
```

図 13 配列アクセスにおけるトランザクション定義

7. 評価

本章では，ソフトウェアキャッシュを実装した DTM システムを評価した結果を示す．

7.1 評価に使用するプログラム

システムの性能評価には，自作のマイクロベンチマーク，および，TM の研究で広く用いられる STAMP [29] から Vacation を使用した．

7.1.1 配列アクセス

まず，配列に対する操作を行うマイクロベンチマークを作成した．このプログラムでは，ノード間で共有する配列に対し，Tx 内で 5 回の読み出しおよび 1 回の書き込みを行う．なお，自身のノードにマッピングされている配列の要素にアクセスする確率を 90%，他のノードにマッピングされている要素にアクセスする確率を 10%とし，書き込みは自身のノードにマッピングされている要素にのみ発生するものとする．また，アクセスする配列の要素は一様乱数を用いてランダムに決定し，Tx 内で同一の要素に複数回アクセスすることが発生しないように設定してある．

具体的な Tx の定義を**図 13** に示す．まず，2 行目で Tx

Vol.2022-ARC-248 No.46
Vol.2022-SLDM-198 No.46
Vol.2022-EMB-59 No.46
2022/3/11

```
1 for (int i = 0; i < NUM_TX; i++) {
2   switch(action) {
3     case MAKE_RESERVATION: {
4       BEGIN_TX();
5       long t = types[n];
6       switch (t) {
7         case RESERVATION_CAR:
8           /* レンタカーの予約価格を決定 */
9           break;
10        case RESERVATION_FLIGHT:
11          /* 飛行機便の予約価格を決定 */
12          break;
13        case RESERVATION_ROOM:
14          /* ホテルの予約価格を決定 */
15          break;
16        default:
17          assert(0);
18      }
19      if (isFound) {
20        /* 顧客情報を登録 */
21      }
22      if (t == RESERVATION_CAR && isFound) {
23        /* レンタカーを予約 */
24      }
25      if (t == RESERVATION_FLIGHT && isFound)
        {
26        /* 飛行機便を予約 */
27      }
28      if (t == RESERVATION_ROOM && isFound) {
29        /* ホテルを予約 */
30      }
31      END_TX();
32    }
33    : /* その他のcase */
34  }
35 }
```

図 14 Vacation におけるトランザクション定義

が開始されると，6 行目に示す rand100() により，0 から 99 の範囲で乱数を生成し，変数 locality との大小比較を行うことで，自身のノードにマッピングされている配列にアクセスするか，他のノードにマッピングされている配列にアクセスするかを決定する．今回の評価では，locality を 90 とする．その後，7 行目または 9 行目に示す rand_local() または rand_remote() により，アクセスする配列の要素を決定する．そして，11 行目に示す TX_READ() により，配列の要素を読み出す．これらの操作を，変数 read_only の回数だけ繰り返す．今回の評価では，read_only を 4 とする．for 文を抜けると，13 行目で同様にアクセスする要素を決定し，14 行目で読み出しを行う．その後，読み出した値を 15 行目でインクリメントし，16 行目に示す TX_WRITE() により，インクリメントした値を配列の要素に書き込む．

7.1.2 Vacation

次に，TM の研究で広く用いられる STAMP から Vacation と呼ばれるプログラムを抜粋した．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．

```
1 template <class KeyType, class ValueType>
2 struct node {
3   KeyType key;
4   ValueType *valuePtr;
5   struct node *parent;
6   struct node *left;
7   struct node *right;
8   long color;
9 };
```

図 15 従来の赤黒木の定義

```
1 template <class KeyType, class ValueType>
2 struct node {
3   KeyType key;
4   global_ptr<ValueType> valuePtr;
5   global_ptr<node> parent;
6   global_ptr<node> left;
7   global_ptr<node> right;
8   long color;
9 };
```

図 16 書き換え後の赤黒木の定義

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 14** に示す．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，19 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の登録を行う．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，22 行目から 30 行目の if 文のうち 1 つが実行され，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，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．

なお，STAMP はマルチコアプロセッサ向け TM のベンチマークであり，そのままでは提案システムで実行することはできないため，移植作業を行う必要がある．Vacation では，データベースが赤黒木で実装されており，これをノード間で共有できるように，赤黒木の定義を書き換える．従来の STAMP における赤黒木の定義を**図 15** に，提案システムで動作するように書き換えた後の赤黒木の定義を**図 16** に示す．図 15 に示すように，STAMP では赤黒木の節点を構造体として定義している．メンバ変数のうち，4 行目から 7 行目に示すポインタを，図 16 に示すように，提案システムにおける共有オブジェクトを表す global_ptr<>に書き換えた．提案システムでは，このような軽微な変更だけで，計算ノード間で共有する赤黒木を定義できる．

Vol.2022-ARC-248 No.46
Vol.2022-SLDM-198 No.46
Vol.2022-EMB-59 No.46
2022/3/11

表 1　評価環境

OS	CentOS 7.5.1804
Processor	Intel Core i5-7500
Clock	3.40 GHz
Physical/Logical #cores	4/4 cores
L1-dcache	32 KiBytes
L2-cache	256 KiBytes
L3-cache	6144 KiBytes
Memory	8 GiBytes
Compiler	g++ 7.1.0

7.2　評価結果

7.1 節で述べた二種類のベンチマークプログラムを用いて，提案システムを評価する．評価には**表 1** に示す計算ノードを計 16 台使用し，これらを 1Gbps のイーサネットに接続した環境を使用した．評価結果を**図 17** および**図 18** に示す．図では，各プログラムの実行結果を折れ線で示している．また，グラフの縦軸は実行時間を表し，横軸は計算ノード数を表している．

まず，配列アクセスでは，2 ノードで実行した場合に 1 ノードで実行した場合よりも実行時間が大きく増加してしまったものの，4 ノードで実行した場合は同等の実行時間まで高速化できている．さらに，8 ノードで実行した場合は 1 ノードよりも実行時間を削減できており，1.97 倍の速度向上を達成した．また，プロトタイプと比較すると，1 ノードで実行した場合の実行時間は同等であり，2 ノード以上で実行した場合は，いずれのノード数においてもプロトタイプの性能を上回っていることがわかる．なお，16 ノードで実行した場合，プロトタイプおよび提案手法のどちらにおいても，8 ノードで実行した場合よりも実行時間が悪化してしまった．しかし，プロトタイプでは 2 ノードで実行した場合よりもさらに悪化しているのに対し，提案手法では実行時間の増加を抑えられていることがわかる．最も実行時間の短い 8 ノードでは，プロトタイプと比較して 1.56 倍の性能向上を達成した．次に，Vacation では，1 ノードで実行した場合よりも実行時間を削減することはできなかった．しかし，2 ノード以上で実行した場合の結果にのみ着目すると，8 ノードまでは実行時間を削減できており，プロトタイプよりも性能が改善していることがわかる．また，配列アクセスと同様に，16 ノードで実行した場合に 8 ノードで実行した場合よりも実行時間が悪化しているが，提案手法では実行時間の増加を抑えられている．8 ノードで実行した場合，プロトタイプと比較して 3.64 倍の性能向上を達成した．

次に，各プログラムを 8 ノードで実行した場合におけるソフトウェアキャッシュのキャッシュヒット率を**表 2** に，プロトタイプおよび提案手法の通信回数比とその内訳を**図 19** に示す．図では，各プログラムの通信回数比を 2 本のバーで示しており，左がプロトタイプ，右が提案手法に対応している．また，プロトタイプの結果を 1 として正規化している．図中の凡例は通信回数の内訳を示しており，それぞれ以下の通りである．なお，RemoteValidation はプロトタイプでのみ発生する通信であり，SharedRequest および InvalidRequest は提案手法でのみ発生する通信である．

RemoteRead:　読み出しに要した通信
RemoteWrite:　書き込み（ロック獲得）に要した

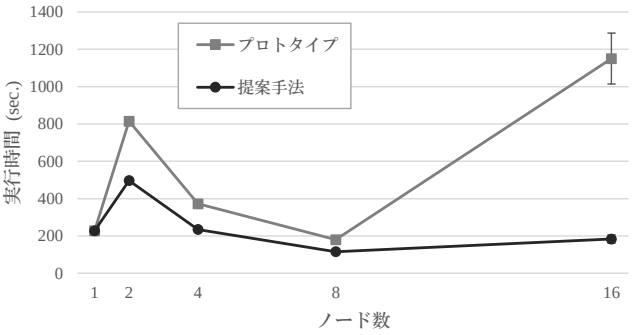


図 17　配列アクセスの評価結果

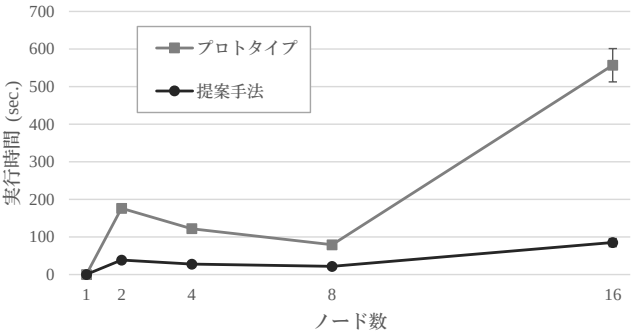


図 18　Vacation の評価結果

表 2　ソフトウェアキャッシュのキャッシュヒット率

	配列アクセス	Vacation
参照数	420,164.9	425,401.0
ヒット数	22,846.4	361,599.1
ミス数	397,318.5	63,801.9
ヒット率（%）	5.4	85.0

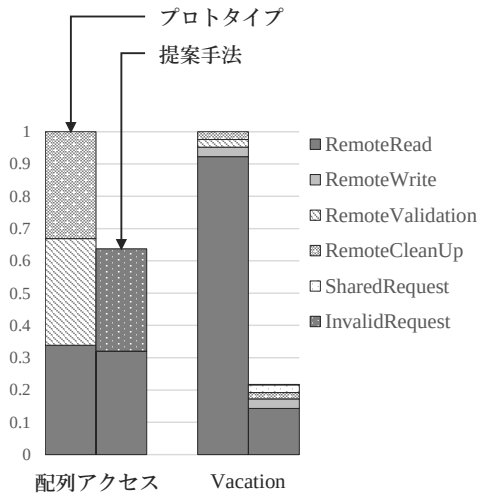


図 19　通信回数比

通信

RemoteValidation: Validation に要した通信

RemoteCleanUp: ロック解放および，プロトタイプでは Validation に必要なメタデータのクリアに要した通信，提案手法ではグローバルメモリへの書き戻しに要した通信

SharedRequest: Shared 状態への状態遷移のリクエストに要した通信

InvalidRequest: Invalid 状態への状態遷移のリクエストに要した通信

7.3 考察

まず，配列アクセスでは，表 2 よりキャッシュヒット率が 5.4%と非常に低いことがわかる．これは，一様乱数を用いてアクセスする配列の要素を決定しているため，同一の要素に連続してアクセスする可能性が低かったからだと考えられる．また，他のノードにマッピングされている要素に対する読み出しが発生する確率は 10%であり，書き込みを自身のノードにマッピングされている要素に限定していることから，キャッシュした要素を再利用する前に，その要素がマッピングされているノードで実行される Tx により書き込みが発生し，キャッシュした要素が無効化されてしまう可能性が高かったと考えられる．キャッシュヒット率が低いにも関わらずプロトタイプと比較して性能が向上した要因として，競合検出にかかる通信回数を削減できたことが考えられる．図 19 より，キャッシュヒット率が低いことから RemoteRead は 5.5%の削減に留まっているが，全体では通信回数を 36.3%削減できている．プロトタイプでは，他のノードにマッピングされているアドレスに対する読み出しが発生した場合，Validation およびそれに使用したメタデータをクリアするための通信が発生する．これに対し，提案手法では，Validation が不要であるため，これらの通信が発生しない．代わりに Invalid 状態への状態遷移のリクエストによる通信が発生するが，Validation とは異なり，各ノードに対して一回の通信で済む．これにより，競合検出にかかる通信回数を 52.1%削減できたことで，プロトタイプよりも性能が向上したと考えられる．

次に，Vacation では，表 2 よりキャッシュヒット率が 85.0%であり，キャッシュが有効に働いていることがわかる．これは，Vacation は読み出しが多く書き込みが少ないプログラムであることが要因だと考えられる．Vacation では，赤黒木で実装されたデータベースを検索する際に多くの読み出しが発生し，このときにソフトウェアキャッシュに値をキャッシュする．一方で，データベースに対する書き込みが少ないため，キャッシュした値が無効化されにくい．このことから，Vacation はソフトウェアキャッシュを活用しやすいプログラムであり，キャッシュヒット率が高くなったと考えられる．実際に，図 19 に示すグラフより，プロトタイプと比較して RemoteRead を 74.6%削減できている．これにより，プロトタイプよりも性能が大幅に向上したと考えられる．

一方で，提案手法においても，単一ノードで逐次実行した場合よりも速度を向上させることができなかった．これは，STAMP がマルチコアプロセッサ向け TM を前提としたプログラムであり，分散並列化することを想定していないプログラムであるからだと考えられる．単一ノードでの実行でも長い時間を要しないプログラムをマルチノードで実行したため，実行時間における通信時間の占める割合が大きくなり，速度が向上しなかったと考えられる．この評価結果において重要な点は，7.1.2 項で示した軽微な変更だけで，マルチコアプロセッサ向けのプログラムからマルチノード向けのプログラムを作成でき，それがノード数によらず正常に動作したということである．この「マルチノード向けのプログラムが容易に記述でき，それが正常に動作する」という性質は，様々なアプリケーションの開発初期段階において非常に重要である．

7.4 さらなる高速化のための方針

本稿では，各ノードにおいて Tx を実行するスレッドが 1 つであることを前提としてソフトウェアキャッシュを設計してきた．しかし，近年はプロセッサ内のコア数が増大しているため，Tx を実行するスレッドが 1 つだけでは，これらのコアを有効に活用しきれていない．そのため，ノード内で複数のスレッドが Tx を実行できるようにシステムを拡張していくことで，さらなる性能向上が期待できる．

複数のスレッドが Tx を実行する場合，ソフトウェアキャッシュをスレッドごとに実装する方式と，各ノードに一つだけ実装し，スレッド間で共有する方式が考えられる．スレッドごとに実装する場合，それぞれのソフトウェアキャッシュにアクセスするスレッドは当該スレッドおよび *Agent Thread* だけとなるため，ソフトウェアキャッシュの制御が単純になるという利点がある．一方で，あるスレッドがキャッシュした値を，他のスレッドが利用することができず，スレッドごとに他のノードから読み出す必要があるため，通信回数が増加する可能性がある．また，これにより同一アドレスの値のコピーがノード内のメモリ上に最大でスレッド数と同数存在することになり，メモリの利用効率が低下する．これに対し，スレッド間で共有する場合，複数のスレッドが同時にソフトウェアキャッシュに対してアクセスした際に，一貫性のない値を読み出してしまうことを防ぐ必要があるため，キャッシュの制御が複雑になるという欠点がある．一方で，あるスレッドがキャッシュした値を，他のスレッドが利用できるようになるため，キャッシュヒット率が向上する可能性があり，スレッドごとに用意する場合と比較して通信回数を削減できる可能性

がある．そのため，通信回数を削減するという観点から，各ノードにソフトウェアキャッシュを一つだけ実装し，スレッド間で共有する方式が適していると考えられる．

ソフトウェアキャッシュを共有する場合，ノード内のスレッド間で排他制御が必要になる処理が増大すると考えられる．現在の設計でも，Tx を実行するスレッドと *Agent Thread* が同時に同一のキャッシュラインに対してアクセスすることを防ぐため，排他制御が必要な処理が存在しているが，Tx を実行するスレッド数を増やすことで，ソフトウェアによる並行性制御のコストが大きくなり，性能に悪影響を与える可能性がある．そこで，これらの制御に HTM を活用することを検討する．ノード内で行われるソフトウェアキャッシュの制御の一部を，HTM で実行する Tx として定義することで，ハードウェアにより競合を検出できるため，高速な制御が期待できる．2.2 節で述べたように，HTM が備える記憶領域で記憶できるアクセス数には上限があるが，ソフトウェアキャッシュの制御であれば，HTM で実行する処理は比較的少なく，上限を超えてしまう可能性は低いと考えられる．したがって，ノード間の一貫性制御にはソフトウェアで実装された DTM を使用し，ノード内の一貫性制御にはハードウェアで実装された HTM を活用することで，システムのさらなる性能向上に繋がる可能性がある．

8. 結論

本稿では，DTM に適したソフトウェアキャッシュを設計および実装した．ソフトウェアキャッシュの設計にあたり，実プロセッサの HTM を参考にした．実プロセッサの HTM は，ハードウェアキャッシュおよびコヒーレンスプロトコルを用いて実装されていることから，この動作をソフトウェアでエミュレートするという方針をとった．また，ソフトウェアキャッシュを用いて，マルチコアプロセッサ向け STM の問題点であった競合検出のオーバヘッドを削減する方法についても提案した．

実装したソフトウェアキャッシュの有効性を検証するため，マイクロベンチマークおよび STAMP ベンチマークから抜粋したプログラムでシステムを評価した．評価の結果，プロトタイプと比較して，マイクロベンチマークでは 1.56 倍，STAMP では 3.64 倍の性能向上を達成した．また，マルチコアプロセッサ向けのプログラムから，軽微な変更でマルチノード向けプログラムを作成でき，正常に動作することを確認した．

今後の展望として，各ノードにおいて複数のスレッドが Tx を実行できるように拡張し，ノード内で行われるソフトウェアキャッシュの制御に HTM を活用することで，システムをさらに高速化することが挙げられる．また，評価結果より，16 ノードで実行した場合に 8 ノードで実行した場合よりも実行時間が悪化してしまったため，この原因について詳細に調査し，より多くのノード数で実行した場合でも性能が向上する方法について検討していくことが挙げられる．

謝辞 本研究の一部は，JSPS 科研費 21H03408 の助成を受けたものである．

参考文献

[1] Yokokawa, M., Shoji, F., Uno, A., Kurokawa, M. and Watanabe, T.: The K computer : Japanese next-generation supercomputer development project, *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, pp. 371–372 (online), available from ⟨https://academic.microsoft.com/paper/2005129866⟩ (2011).

[2] RIKEN Center for Computational Science: Supercomputer Fugaku, `https://www.r-ccs.riken.jp/en/fugaku/`.

[3] Protic, J., Tomasevic, M. and Milutinovic, V.: Distributed shared memory: concepts and systems, *IEEE Parallel & Distributed Technology: Systems & Applications*, Vol. 4, No. 2, pp. 63–79 (online), available from ⟨https://academic.microsoft.com/paper/2097219246⟩ (1996).

[4] Bisiani, R. and Ravishankar, M.: PLUS: a distributed shared-memory system, *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pp. 115–124 (online), DOI: 10.1109/ISCA.1990.134514 (1990).

[5] Li, K.: IVY: a shared virtual memory system for parallel computing, *Proceedings of the International Conference on Parallel Processing*, Vol. 2, pp. 94–101 (online), available from ⟨https://academic.microsoft.com/paper/51427095⟩ (1988).

[6] Bennett, J. K., Carter, J. B. and Zwaenepoel, W.: Munin: Distributed shared memory based on type-specific memory coherence, *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pp. 168–176 (1990).

[7] PGAS Forum: PGAS Partitioned Global Address Space, `http:/www.pgas.org/`.

[8] Chamberlain, B., Callahan, D. and Zima, H.: Parallel Programmability and the Chapel Language, *ieee international conference on high performance computing data and analytics*, Vol. 21, No. 3, pp. 291–312 (online), available from ⟨https://academic.microsoft.com/paper/2090409324⟩ (2007).

[9] Numrich, R. W. and Reid, J.: Co-array Fortran for parallel programming, *ACM Sigplan Fortran Forum*, Vol. 17, No. 2, pp. 1–31 (online), available from ⟨https://academic.microsoft.com/paper/2140300123⟩ (1998).

[10] Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H. and Apr, E.: Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit, *ieee international conference on high performance computing data and analytics*, Vol. 20, No. 2, pp. 203–231 (online), available from ⟨https://academic.microsoft.com/paper/2095760405⟩ (2006).

[11] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze,

L., Kahan, S. and Oskin, M.: Latency-tolerant software distributed shared memory, *USENIX ATC '15 Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, pp. 291–305 (online), available from ⟨https://academic.microsoft.com/paper/2135875530⟩ (2015).

[12] El-Ghazawi, T. and Cantonnet, F.: UPC Performance and Potential: A NPB Experimental Study, *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pp. 1–26 (online), available from ⟨https://academic.microsoft.com/paper/2109217687⟩ (2002).

[13] Bachan, J., Baden, S. B., Hofmeyr, S., Jacquelin, M., Kamil, A., Bonachea, D., Hargrove, P. H. and Ahmed, H.: UPC++: A high-performance communication framework for asynchronous computation, *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, pp. 963–973 (2019).

[14] Herlihy, M. and Moss, J. E. B.: Transactional memory: Architectural support for lock-free data structures, *Proceedings of the 20th annual international symposium on Computer architecture*, pp. 289–300 (1993).

[15] 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).

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

[17] International Business Machines Corporation: Power ISA® Version 2.07, `https://www.power.org/documentation/power-isa-version-2-07/` (2013).

[18] Dice, D., Shalev, O. and Shavit, N.: Transactional locking II, *International Symposium on Distributed Computing*, Springer, pp. 194–208 (2006).

[19] Felber, P., Fetzer, C. and Riegel, T.: Dynamic performance tuning of word-based software transactional memory, *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 237–246 (online), available from ⟨https://academic.microsoft.com/paper/2149089882⟩ (2008).

[20] Dalessandro, L., Spear, M. F. and Scott, M. L.: NOrec: streamlining STM by abolishing ownership records, Vol. 45, No. 5, pp. 67–78 (online), available from ⟨https://academic.microsoft.com/paper/2155500238⟩ (2010).

[21] Couceiro, M., Romano, P., Carvalho, N. and Rodrigues, L.: D2STM: Dependable distributed software transactional memory, *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, IEEE, pp. 307–313 (2009).

[22] Saad, M. M. and Ravindran, B.: Hyflow: A high performance distributed software transactional memory framework, *Proceedings of the 20th international symposium on High performance distributed computing*, pp. 265–266 (2011).

[23] Siek, K. and Wojciechowski, P. T.: Atomic RMI: A distributed transactional memory framework, *International Journal of Parallel Programming*, Vol. 44, No. 3, pp. 598–619 (2016).

[24] Bocchino, R. L., Adve, V. S. and Chamberlain, B. L.: Software transactional memory for large scale clusters, *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 247–258 (2008).

[25] Papamarcos, M. S. and Patel, J. H.: A low-overhead coherence solution for multiprocessors with private cache memories, *Proceedings of the 11th annual international symposium on Computer architecture*, pp. 348–354 (1984).

[26] Thomadakis, M. E.: The architecture of the Nehalem processor and Nehalem-EP SMP platforms, *Resource*, Vol. 3, No. 2, pp. 30–32 (2011).

[27] Advanced Micro Devices: AMD64 Architecture Programmer' s Manual (2021).

[28] Le, H. Q., Guthrie, G. L., Williams, D. E., Michael, M. M., Frey, B. G., Starke, W. J., May, C., Odaira, R. and Nakaike, T.: Transactional memory support in the IBM POWER8 processor, *IBM Journal of Research and Development*, Vol. 59, No. 1, pp. 8–1 (2015).

[29] 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).