情報処理学会論文誌　データベース　Vol.10　No.2　19–30 (June 2017)

RDMAの適用による RAMPトランザクション処理の高速化

村田 直郁[1,a)]　川島 英之[2]　建部 修見[2]

受付日 2016年12月10日, 採録日 2017年2月8日

概要：分散データベース管理システムにおいて外部キー制約や二次索引，実体化ビューの管理を行うための高性能な処理方式として Read Atomic Multi-Partition（RAMP）トランザクションがある．RAMP トランザクションは隔離性を緩和することで高性能化されているが，それを先進的デバイスによって高性能化する技法は未開拓である．そこで，本研究では高性能インターコネクトである InfiniBand を利用し，Remote Direct Memory Access（RDMA）の機能を用いて RAMP トランザクションを高速化する手法を提案する．まず，RDMA-Write による GET/PUT オペレーションの高速化手法として GET+/PUT+方式を提案する．続いて，RDMA-Read によるさらなる GET オペレーションの高速化手法として GET*方式を提案する．提案手法の評価のため，プロトタイプ In-Memory Key-Value Store を実装する．Yahoo! Cloud Serving Benchmark を用いた実験において，従来方式と比べて最大 2.67 倍の高速化を達成することを示す．

キーワード：分散トランザクション，RAMP トランザクション，InfiniBand，RDMA

Accelerating RAMP Transaction with RDMA

Naofumi Murata[1,a)]　Hideyuki Kawasima[2]　Osamu Tatebe[2]

Received: December 10, 2016, Accepted: February 8, 2017

Abstract: RAMP transaction demonstrates high efficiency for the management of foreign key constraints, secondary indices, and materialized views in distributed environment. RAMP transaction realizes high performance distributed transaction processing by relaxing isolation level but it is not considered to accelerating performance by using fast communication method. This paper presents the acceleration of RAMP transaction exploiting Remote Direct Memory Access (RDMA) on InfiniBand. We first present GET+ and PUT+ operations that accelerate RAMP transaction exploiting RDMA-Write. Then we present GET* operation that further accelerates GET+ operation exploiting RDMA-Read. To evaluate the proposed methods, we implement a prototype in-memory key value store in C/C++. The results of experiments show that compared with RAMP transaction on IP over InfiniBand which emulates IP network on InfiniBand, GET*/PUT+ demonstrate 2.67x performance improvement with Yahoo! Cloud Serving Benchmark.

Keywords: distributed transaction, RAMP transaction, InfiniBand, RDMA

1. はじめに

人類が扱うデータ量は年々爆発的に増加し続けており [1]，それらを高速に処理するシステムとして，分散データベース管理システム（分散 DBMS）が活発に研究・開発されてきている [2], [3], [4], [5]．分散 DBMS では，複数ノードにデータを分割して管理することでスケーラビリティや耐障害性を提供できる反面，複数ノードにまたがるトランザクション処理は非常に高コストとなるため，その高性能化が重要な課題となっている．分散 DBMS において外部キー制約や二次索引，実体化ビューの管理を行う場合，

[1] 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[2] 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[a)] murata@hpcs.cs.tsukuba.ac.jp

複数ノードに配置されたデータを同時に更新する必要がある．このような処理を実行する場合，2-Phase Locking（2PL）プロトコルと呼ばれる方式を用いて，更新するアイテムのロックを取得し，排他制御を行うことで正確な処理が実現されている．一方，分散環境では 2PL はコストが高いため，実際のアプリケーションに適応することが難しいという問題がある [6]．この問題に対して Bailis らはまず，新たな隔離性レベルとして **Read Atomic 隔離性**を定義した [6]．Read Atomic 隔離性は Read Committed 隔離性よりも強い隔離性レベルであり，外部キー制約や二次索引，実体化ビューの管理に求められるレベルを満足している．また，Facebook における「いいね！」の処理や LinkedIn におけるメールボックスの管理など，現実的なアプリケーションが多く存在する実用的な隔離性レベルである．さらに Bailis らは，Read Atomic 隔離性を保証しつつ 2PL よりも高性能な処理方式として **Read Atomic Multi-Partition（RAMP）トランザクション**を提案した [6]．RAMP トランザクションはマルチバージョンに基づく効率的なプロトコルにより，ロックを使わずに Read Atomic 隔離性を保証でき，高いスループットとスケーラビリティを実現している．

RAMP トランザクションは隔離性レベルを緩和することで高性能化されているが，それを先進的デバイスによって高性能化する技法については考えられていない．そこで本研究では，高性能インターコネクトである InfiniBand [7] を用いて RAMP トランザクションの高速化を図る．なお，本研究における高速化は，単一トランザクションの応答時間（レイテンシ）の短縮ではなく，単位時間あたりに処理できるトランザクション数（スループット）の向上を目指すものである．RAMP トランザクションには RAMP-Fast，RAMP-Small，RAMP-Hybrid の 3 つのアルゴリズムがあるが，本研究では，ほとんどのケースで最も高速に処理を終えることができる **RAMP-Fast** アルゴリズムのみを対象とする．本論文では，“RAMP トランザクション” という表記は RAMP-Fast アルゴリズムを指すものとする．

単純に InfiniBand のソケットインタフェースを利用すれば，RAMP トランザクションを高速化できることは明らかであるが，**Remote Direct Memory Access（RDMA）**を用いることでさらなる高速化が可能となる．RDMA とは，リモートノードのメモリ上にあるデータに対して，リモート CPU の介在なしにデータを読み書きするネットワーク技術である．RDMA の大きな特徴として，データをカーネルバッファにコピーすることなく転送できるということがあげられる．この特徴により，高速なデータ転送を実現できる反面，通信にリモート CPU が介在しないため独自の通信プロトコルを設計する必要がある．

本研究ではまず，RDMA-Write を用いた RAMP トランザクションにおける GET/PUT オペレーションの高速化手法として GET+/PUT+方式を提案する．GET+/PUT+方式では，クライアントは RDMA-Write を用いてリクエストをサーバのメッセージバッファに書き込む．サーバはポーリングによって，メッセージバッファからリクエストを取得し処理した後，結果を RDMA-Write でクライアントのメッセージバッファに書き込む．クライアントはポーリングによって，メッセージバッファから結果を取得する．この方式により確かに RAMP トランザクションを高速化できるが，サーバがつねにリクエストの受付・テーブル探索・結果の送信を行う必要があり，クライアントはその間待たされるため，そこが性能劣化につながる可能性がある．

そこで，サーバにおいてこれらの処理を行う割合を抑えたさらなる高速化手法として GET*方式を提案する．GET*方式はクライアントが RDMA-Read を用いて，サーバのメモリ上にあるアイテムを直接読み出すことで高速化を実現する．この方式では，サーバ側でテーブル探索を行わずにクライアントのみでアイテムを取得できるため，GET+方式よりも効率的である一方，他のトランザクションによって更新中のアイテムを読み出してしまうという問題が発生する．そこで我々はサーバ側においてアイテムに無効化ビットを与える．もし，クライアントが読み出したアイテムが無効だった場合，GET+方式に切り替えてアイテムの再取得を行う．また，GET*方式を実行するためには，クライアントはサーバのメモリのどこに所望のアイテムが存在するか知る必要がある．そこでクライアント側にアドレスキャッシュを用意する．Facebook の報告 [8] によれば，実際のワークロードにおける read の割合は非常に高く，GET オペレーションの高速化は有益であると考えられる．

提案手法の評価のため，C/C++言語を用いてプロトタイプ In-Memory Key-Value Store を実装し，従来手法との比較実験を行う．実験では，従来手法として Strict 2PL と RAMP トランザクションを実行する．従来手法の通信には，InfiniBand 上で IP 通信を行う IP over InfiniBand（IPoIB）を利用する．提案手法として，GET+と PUT+方式を適用した RAMP トランザクションと，GET*方式と PUT+方式を適用した RAMP トランザクションを実行する．実験には自作のマイクロベンチマークと Yahoo! Cloud Serving Benchmark（YCSB）[9] を用いる．

本論文の構成は以下のとおりである．2 章では，本研究の研究背景について解説する．3 章では，本研究の提案手法について解説する．4 章では，提案手法の評価のため実装したプロトタイプ In-Memory Key-Value Store の設計と実装について解説する．5 章では，提案手法の評価のため行った実験の概要と結果について解説する．6 章では，結論を述べる．

2. 研究背景

2.1 Key-Value Store

今日，各種データ処理の基盤として Oracle [10] や PostgreSQL [11], MySQL [12] に代表されるリレーショナルデータベース管理システム（RDBMS）が広く利用されている．RDBMS は，リレーショナルデータモデルに基づくデータの構造化や SQL による複雑な検索処理，トランザクション処理など，データの一貫性を保ちつつ複雑な処理も提供できることから，様々な分野でデータ分析処理に利用されている．しかし，データ量の増加にともない，データを分散させるケースが増えてくるに従って，RDBMS の利点である複雑な処理や一貫性の保証などがボトルネックとなり，性能が劣化してしまうという問題がある．このような問題から，分散 DBMS ではより柔軟にデータを扱うことができるデータモデルを採用し，データに対する操作もよりシンプルなものを提供するという設計が一般的となっている．代表的なデータモデルには，データを key と value のペアで管理する **Key-Value 型**があり，このデータモデルに基づく分散 DBMS を**分散 Key-Value Store**（**KVS**）と呼ぶ．代表的な KVS には Google の Bigtable [2] や Amazon の Dynamo [3]，Basho の Riak [13] などがあげられる．KVS では，データに対する読み込み/書き込み操作として，**get/put オペレーション**[*1]を提供している．get オペレーションは key の値を指定することでテーブルから対応する value の値を取得し，put オペレーションは key と value の値を両方指定して value の値を更新する．本研究では，分散 DBMS として KVS を対象とし，KVS は get/put オペレーションのみを提供するものと仮定する．

2.2 RAMP トランザクション

分散 DBMS において，外部キー制約や二次索引，実体化ビューの管理を行うためには，複数ノードのデータを分散トランザクションによって同時に更新する必要がある．このとき，正確な処理を実現するためにトランザクションが満たすべき性質として，“トランザクションの各更新が他のトランザクションからすべて観測されるか，まったく観測されないかのどちらかでなければならない” という性質がある．例として，初期値がともに 0 のアイテム x と y に対して，あるトランザクションが x と y の値をともに 1 に更新する場合，他のトランザクションからは $x = 1$，$y = 1$ もしくは $x = 0$，$y = 0$ のどちらかで読み出されなければならない．このとき，$x = 1$，$y = 0$ もしくは $x = 0$，$y = 1$ という結果を読み出してしまった場合，一部古いバージョンを読み出しており，トランザクションの更新を部分的にしか読み出せていないことになる．この不整合を **fractured read** と呼び，これを防ぐ新たな隔離性レベルとして Read Atomic 隔離性が定義された [6]．Read Atomic 隔離性は，Facebook における「いいね！」の処理や LinkedIn におけるメールボックスの管理など，現実的なアプリケーションが多く存在する実用的な隔離性レベルである．

Read Atomic 隔離性はロックを用いた排他制御によって保証できるが，ロックはコストが高いため，新たな処理方式として Read Atomic Multi-Partition（RAMP）トランザクションが提案された [6]．RAMP トランザクションはロックを使わず，Read Atomic 隔離性を保証でき，高いスループットとスケーラビリティを実現している．通常，分散トランザクションは 2 フェーズで実行される [14] が，RAMP トランザクションにおける read は理想的な状況下では 1 フェーズで実行できる．以下では，RAMP トランザクションにおける書き込み/読み込み操作である PUT/GET オペレーションについて解説する．

PUT オペレーションは複数アイテムに対する put オペレーションを実現するための操作であり，*prepare*，*commit* の 2 フェーズで実行される．各トランザクションにはタイムスタンプが割り当てられ，更新するアイテムに対してタイムスタンプを用いて新たなバージョンを生成する．*prepare* フェーズでは，クライアントはアイテムの新しいバージョンとあわせて，メタデータとしてトランザクションによって更新される他のアイテムの一覧を送信する．サーバは送られたアイテムとメタデータをテーブルに追加する．*commit* フェーズでは，クライアントはサーバにトランザクションのタイムスタンプを送る．サーバは送られたタイムスタンプを用いてコミット処理を実行する．

GET オペレーションは複数アイテムに対する読み込み操作であり，理想的な状況下では 1 フェーズで実行される．クライアントはまず，get オペレーションを複数実行し，各アイテムのコミット済みの最新バージョンを読み出す．このとき，そのアイテムに対して PUT オペレーションを実行しているトランザクションがほかにいなければ，処理を終えることができる．しかし，PUT オペレーションとの競合により fractured read が発生している場合，それを検知し，正しいバージョンの再取得を行う必要がある．そこで GET オペレーションでは，アイテムとあわせて格納されているメタデータを用いて fractured read を検知し，各アイテムの正しいバージョンを特定する．fractured read が検知された場合，クライアントはサーバに対し，タイムスタンプを指定してアイテムの正しいバージョンを取得する．

RAMP トランザクションには，通信回数やデータサイズの異なる 3 つアルゴリズム：RAMP-Fast，RAMP-Small，RAMP-Hybrid があるが，本研究では，ほとんどのケースで最も高速に処理を終えることができる RAMP-Fast アル

[*1] 本論文では，KVS における単一アイテムに対する読み込み/書き込み操作を get/put オペレーションと表記し，RAMP トランザクションにおける複数アイテムに対する読み込み/書き込み操作を GET/PUT オペレーションと表記する．

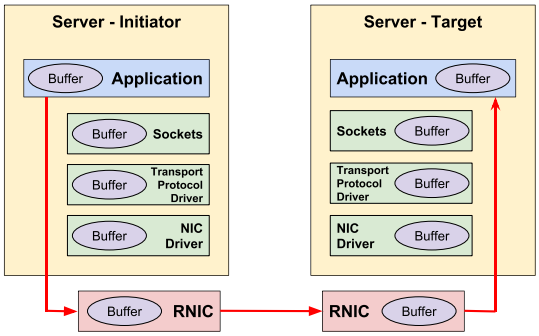

図 1　RDMA zero-copy interconnect（文献 [20] より引用）
Fig. 1　RDMA zero-copy interconnect (cited from [20]).

ゴリズムのみを対象とする．

2.3　InfiniBand を用いた通信

近年，HPC の分野ではサーバ間をつなぐインターコネクトとして，従来の Ethernet よりも高性能なインターコネクトが広く採用されるようになった．InfiniBand は現在，最も高いシェアを誇る [15] 高性能インターコネクトであり，従来の Ethernet に比べて高スループット・低レイテンシの通信を実現できる．また，年々低価格化が進んでいる [16] ことから一般企業での採用事例 [17], [18], [19] も増えており，今後幅広く普及していくことが予想される．

InfiniBand における通信の大きな特徴として，データをカーネルバッファにコピーすることなく転送する，**zero-copy 通信**が可能であることがあげられる．zero-copy 通信におけるデータ転送の流れを**図 1** に示す．ローカル側は送信したいデータのアドレスを指定して送信要求を発行し，リモート側では受信先のアドレスを指定して受信要求を発行する．データはカーネルバッファにコピーされることなく NIC 内のバッファにコピーされ，転送される．リモート側でも同様にカーネルバッファを経由せずに指定したアドレスに書き込まれる．zero-copy 通信を実現するためには，NIC がユーザプログラムの仮想アドレス空間を理解できる必要がある．したがって，zero-copy 通信を行う際は，あらかじめデータ送受信に用いるメモリ領域を OS に登録しておき，仮想メモリアドレスと物理メモリアドレスの変換テーブルを作成して NIC に渡しておく．この機構により，InfiniBand ではリモートのメモリアドレスを指定して直接データにアクセスする **Remote Direct Memory Access（RDMA）**の機能を利用できる．

InfiniBand 上で API を用いて zero-copy 通信を行う方式にはいくつか種類があるが，ここではその中で最も代表的な 3 つの方式について解説する．

1 つめの方式は **Send/Recv-Verbs** である．この方式は Send/Recv オペレーションによってデータを送受信する基本的な通信方式である．ローカル側は送信したいデータのアドレスを指定し，Send オペレーションを実行する．一方，リモート側は受信先のアドレスを指定し，Recv オペレーションを実行する．このとき，データはカーネルバッファを経由せずに転送されるため，ソケットインタフェースを利用するよりも高速にデータ転送を行うことができる．

2 つめの方式は **RDMA-Write** である．この方式は RDMA を行う方式の 1 つであり，リモートのメモリ領域を指定して直接データを書き込む方式である．ローカル側は書き込みたいリモートのメモリ領域を指定し，Send オペレーションを実行する．一方，リモート側は Recv オペレーションを実行する必要はなく，ローカル側が指定したリモートのメモリ領域にデータが書き込まれる．したがって，この方式ではカーネルバッファを経由せず，かつリモートの CPU をまったく使わずにデータ転送を行えるため，Send/Recv-Verbs よりも高速にデータ転送を行うことができる．

3 つめの方式は **RDMA-Read** である．この方式は RDMA を行う方式の 1 つであり，リモートのメモリ領域を指定して直接データを読み出す方式である．ローカル側は読み出したいリモートのメモリ領域と受信先のアドレスを指定し，Send オペレーションを実行する．一方，リモート側は Recv オペレーションを実行する必要はなく，ローカル側が指定したリモートのメモリ領域が読み出される．したがって，この方式もカーネルバッファを経由せず，かつリモートの CPU をまったく使わずにデータを転送できるため，Send/Recv-Verbs よりも高速にデータ転送を行える．

Zero-copy 通信とは別に InfiniBand では **IP over InfiniBand（IPoIB）**と呼ばれる通信サービスが提供されている．IPoIB は，InfiniBand ネットワーク上で IP ネットワーク層を構成する通信サービスである．IPoIB を使う利点として，ソケットを使った既存のプログラムからでも InfiniBand を利用することができ，InfiniBand 用にプログラムを修正する必要がないということがある．しかし，IPoIB では zero-copy 通信を行うことができないため，InfiniBand の他の通信方式と比べて性能が制限されてしまうことが知られている [16], [21]．

2.4　関連研究

RDMA による KVS の高性能化に関する研究には，Pilaf [16]，HERD [22]，HydraDB [23] がある．Pilaf は RDMA-Read と Send/Recv-Verbs を用いて高性能な In-Memory KVS を実現した研究である．Pilaf では，get オペレーションを RDMA-Read で実行し，put オペレーションを Send/Recv-Verbs で実行する．get オペレーションではまず，RDMA-Read を用いてハッシュエントリを読み出し，アイテムのアドレスを取得する．そして，取得したアドレスを用いて再度 RDMA-Read を実行し，アイテムを取得する．この get オペレーションは他の put オペレー

ションと競合する可能性があるため，Pilaf ではチェックサムを用いて競合検知を行う．HERD は RDMA-Write と Send/Recv-Verbs を用いて高性能な In-Memory KVS を実現した研究である．HERD では，クライアントは RDMA-Write を用いてリクエストをサーバのリクエストバッファに書き込み，サーバはポーリングによってリクエストを取得し，処理した結果を Send/Recv-Verbs でクライアントに返す．HERD は高速化のために Unreliable なデータ転送サービスを利用しており，データロスの検知や再送制御などがハードウェアレベルで行われない．したがって，それらをアプリケーションレベルで保証する必要がある．一方，本研究では Reliable なデータ転送サービスを利用するため，その問題は解決されている．HydraDB は RDMA-Write/Read を用いて高性能な In-Memory KVS を実現した研究である．本研究では，HydraDB で用いられている 3 つの手法を採用している．1 つは GET+/PUT+ オペレーションにおけるメッセージフォーマット，もう 2 つは GET*オペレーションにおけるアドレスキャッシュと無効化ビットの機構である．HydraDB は RDMA を用いたロギングやレプリケーションも提供している．これらの研究はいずれも RDMA を効果的に用いることで，高性能な KVS を実現した研究であるが，トランザクションはサポートされていない．一方，本研究はトランザクションをサポートしている．

RDMA によるトランザクション処理の高性能化に関する研究には FaRM [24] がある．FaRM は RDMA-Write/Read を用いてクラスタ上で高速なリモートメモリサービスを実現した研究である．FaRM ではクラスタ上のアイテムに対し，そのレプリカも含めて Serializable 隔離性を保証するトランザクションを実行できる．FaRM は洗練されたハードウェア技術を巧みに活用している．RDMA の利用においては，カーネルドライバを実装しシステムの起動時に連続した巨大なページを確保することで，性能劣化の原因である NIC 内で管理されているページテーブルの肥大化を防いでいる．また，DRAM に無停電電源装置（UPS）を取り付けることによって，不揮発性 DRAM を実現している．FaRM は非常に高速なトランザクション処理を実現している一方，本研究で対象としている Read Atomic 隔離性については考えられていない．

RDMA による RDBMS の高性能化に関する研究には cyclo-join [25], [26] がある．Cyclo-join は，RDBMS において負荷の高い処理である類似結合処理を高速化した研究である．Cyclo-join では，2 つのリレーション R と S をそれぞれ R_1 から R_N，S_1 から S_N に分割し，リング状に接続された N 台のノードに分配する．各ノードで結合処理を行った後，S_i を隣接ノードに転送する．これを繰り返すことで結合処理を完了する．各ノードはそれぞれ InfiniBand により接続され，データ転送に RDMA を用いることで，高速な結合処理を実現している．

3. 提案手法：RAMP with RDMA

RAMP トランザクションは Read Atomic 隔離性に対して，マルチバージョンに基づく効率的なプロトコルを設計することで高いスループットとスケーラビリティを実現している．しかし，RAMP トランザクションを先進的デバイスによって高性能化する手法については未開拓であり，RAMP トランザクションに対する InfiniBand および RDMA の適用はいまだ考えられていない．そこで，本研究では InfinBand を用いて RAMP トランザクションの高速化を図る．InfiniBand を利用する最も単純な方式は IPoIB を利用することである．IPoIB を使うことで，既存のソケットを使ったプログラムに修正を加えることなく，InfiniBand の利用が可能となる．しかし，IPoIB では zero-copy 通信を行うことができず，データはつねにカーネルバッファを経由して転送されるため，従来の Ethernet を用いる場合に比べてあまり高速化されないことが知られている [16]．もう 1 つの基本的な通信方式は，Send/Recv-Verbs である．Send/Recv-Verbs では API を用いて zero-copy 通信を行うため，IPoIB を利用するよりも高速なデータ転送を実現できる．しかし，InfiniBand の性能を最大限に活かすためには Send/Recv-Verbs ではまだ不十分である．

InfiniBand 上で最も高速にデータ転送を行える通信方式は，RDMA-Write/Read である．RDMA-Write/Read は Send/Recv-Verbs と同じく，API を用いて zero-copy 通信を行う方式であるが，通信にリモート CPU が介在しないため，より高速なデータ転送を実現できる．この，通信にリモート CPU が介在しないという特徴は，高速なデータ転送を実現する要因である一方，利用するためには独自の通信プロトコルを設計しなければならないという問題を生む．この問題をふまえたうえで，我々はまず，RDMA-Write を用いた高速化手法として GET+/PUT+方式を提案する．GET+/PUT+方式では，RDMA-Write とポーリングを用いてリクエストと結果の受け渡しを行うという自然な手法を用いる．これにより，純粋に RDMA-Write の性能を活かした高速化が実現できる．しかしながら，GET+/PUT+方式ではサーバがつねにリクエストの受付・テーブル探索・結果の送信を行う必要があり，クライアントはその間待たなければならないため，そこが性能劣化につながる可能性がある．そこで，サーバにおいてこれらの処理を行う割合を抑えたさらなる高速化手法として GET*方式を提案する．GET*方式はクライアントが RDMA-Read を用いて，サーバのメモリ上にあるアイテムを直接読み出すという手法を用いる．これにより，純粋な RDMA-Read の性能を活かした高速化に加え，クライアントの待ち時間を削減することが可能となる．

GET*方式は RDMA-Read を用いた高速化手法である

が，RDMA-Read をトランザクション処理に適用する場合，隔離性をどう保証するかが問題となる．サーバはクライアントによる RDMA-Read の実行を検知することができないため，ロックを用いる並行実行制御が使えないという問題がある．しかし，本研究で対象とする RAMP トランザクションの大きな特徴は，ロックを用いる並行実行制御を必要とせず，特に読み出し操作に関して，理想的な状況下では単にコミット済みのアイテムを読み出すだけであるため，RDMA-Read を自然に適用することが可能となっている．RDMA-Read を用いることによって，クライアントはサーバの CPU をまったく介さずにアイテムの取得を行うことができるため，RDMA-Read を用いる方式は RDMA-Write を用いる方式よりも高速にアイテムの取得を行うことができる．しかし，RDMA-Read でサーバのアイテムを取得する場合，事前に所望のアイテムがサーバのメモリ上のどこに存在するかを知らなければならない．そこで，我々はクライアント側でアドレスを保持する address cache なる機構を新たに導入しこの問題を解決した．これらの詳細は 3.2 節で解説する．

3.1　GET+/PUT+方式

GET+/PUT+方式では RDMA-Write を用いてリクエストと結果の受け渡しを行う．クライアントとサーバはそれぞれメッセージバッファを持ち，クライアントは RDMA-Write を用いてリクエストをサーバのメッセージバッファに書き込む．サーバはメッセージバッファをポーリングすることでリクエストの到着を検知する．リクエストの到着を検知する手段として RDMA-Write with Immediate を利用する方法も考えられるが，この方式では，データの到着を検知するためにリモート側で Recv オペレーションを実行する必要があり，RDMA-Write よりも性能が劣ることが知られているため [23]，本研究では RDMA-Write とポーリングを組み合わせる方式を採用する．

メッセージの到着をポーリングによって検知する場合，RDMA-Write によるメッセージの書き込みが完了したことを知るために，どのアドレスを調べればいいのかを正確に知る必要がある．しかし，メッセージのデータサイズは固定長ではないため，そのアドレスがどこであるかを知ることができない．そのため，メッセージの先頭部分に固定長のデータでメッセージのデータサイズを格納するようなメッセージフォーマットを用いる必要がある．これにより，ポーリングの際はまず先頭部分に格納されているメッセージのデータサイズを読み出し，取得した値をもとにメッセージの終端アドレスを計算して調べることができる．このメッセージフォーマットが HydraDB [23] でも用いられているため，本研究でもそれを採用する．**図 2** にフォーマットの概要を示す．なお，メッセージバッファはコネクションごとにそれぞれ独立して用意されるため，メッセージバッファへの書き込みに対する並行実行制御は必要としない．

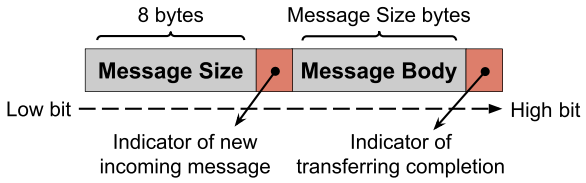

図 2　メッセージフォーマット（文献 [23] より引用）
Fig. 2　Message format (cited from [23]).

クライアントはリクエストをメッセージフォーマットでサーバに送り，サーバはメッセージバッファをポーリングし，リクエストの到着を調べる．メッセージは header と body から構成され，header は 8 バイトの固定長で body のサイズが格納されている．ポーリングではサーバはまず header の到着を検知するために先頭から 9 バイト目にある 1 番目のインジケータを調べる．1 番目のインジケータを調べ，値が更新されていれば先頭 8 バイトの書き込みが完了していることが分かるため header を読み出し，body のサイズを取得する．次に body の到着を検知するため，1 番目のインジケータからさらに body のサイズ分だけスキップした位置にある 2 番目のインジケータを調べる．そのとき，値が更新されていれば body の書き込みが完了していることが分かるため body を読み出し，リクエストの取得が完了する．その後，サーバは取得したリクエストを処理し，メッセージバッファをゼロクリアした後，結果を RDMA-Write を用いてクライアントのメッセージバッファに書き込む．クライアント側でも同様にポーリングを行い，結果の取得を行う．GET+，PUT+方式ともにこの方法に従ってリクエストと結果の受け渡しを行う．

PUT+方式において実行される関数を **Algorithm 1**，**2** に示す．**Algorithm 1** の CLIENT_PUT+関数は PUT+方式において最初に実行される関数であり，各サーバに対して RDMA-Write を用いて PREPARE，COMMIT リクエストを送信する（**Algorithm 1**，line 6–9，line 10–12）．サーバ側では，クライアントから送られてきたリクエストをポーリングによって取得し，リクエストの種類に応じて PREPARE_PUT+，COMMIT_PUT+関数を実行する（**Algorithm 2**）．PREPARE_PUT+関数ではテーブルにバージョンを追加し，アイテムのコミット済みバージョンを無効化する（**Algorithm 2**，line 6–9）．COMMIT_PUT+関数ではコミットされるアイテムのタイムスタンプが，すでにあるコミットされた同じアイテムのタイムスタンプよりも新しい場合，そのアイテムのコミット済みバージョンを更新する（**Algorithm 2**，line 11–19）．

GET+方式も PUT+方式と同様に，RDMA-Write とポーリングを用いてリクエストと結果の受け渡しを行う．サーバ側における get リクエストの処理は GET*方式と同様で

Algorithm 1 PUT+ (Client-Side)

```
 1: payload: message payload with attributes ⟨key, value, timestamp, metadata⟩
 2:
 3: procedure CLIENT_PUT+(W: a set of ⟨key, value⟩)
 4:     ts_tx ← generate a new timestamp
 5:     K_tx ← a set of keys in W
 6:     parallel-for ⟨k, v⟩ ∈ W do
 7:         payload p ← ⟨key = k, value = v, timestamp = ts_tx, metadata = (K_tx − {k}) ⟩
 8:         RDMA-Write ⟨PREPARE, p⟩
 9:     end parallel-for
10:     parallel-for server s that contains a key in W do
11:         RDMA-Write ⟨COMMIT, ts_tx⟩ to s
12:     end parallel-for
13: end procedure
```

Algorithm 2 PUT+ (Server-Side)

```
 1: payload: message payload with attributes ⟨key, value, timestamp, metadata⟩
 2: versions[k]: a set of message payload for key k with version information
 3: latestCommit[k]: lastly committed timestamp for key k
 4: committedVersion[k]: committed version of key k
 5:
 6: procedure PREPARE_PUT+(p: payload)
 7:     versions[p.key].add(p)
 8:     committedVersions[p.key].invalidate
 9: end procedure
10:
11: procedure COMMIT_PUT+(ts_c: timestamp)
12:     K_ts ← {w.key | w ∈ versions ∧ w.timestamp = ts_c}
13:     for k ∈ K_ts do
14:         if latestCommit[k] < ts_c then
15:             latestCommit[k] ← ts_c
16:             committedVersion[k] ← w ∈ versions[k] ∧ w.timestamp = ts_c
17:         end if
18:     end for
19: end procedure
```

あり，3.2 節で GET*方式について述べる．

3.2 GET*方式

GET+方式では，サーバはリクエスト受付・テーブル探索・結果送信を行う．これらの処理が行われている間，クライアントはただ待機しているのみである．この無駄な待機時間を削減する方式が GET*である．GET*方式ではクライアントが RDMA-Read を用いることにより，サーバ CPU の介在なしにサーバのメモリ上にあるアイテムを直接読み出す．したがって，GET*方式は GET+方式でサーバがクライアントに強いる待機時間を削減する．GET*方式において実行される関数を **Algorithm 3**，**4** に示す．

クライアントが RDMA-Read によってサーバのメモリ上にあるアイテムを読み出すためには，そのアイテムのアドレスを知る必要がある．そこで，GET*方式ではクライアント側にアドレスキャッシュを用意する（**Algorithm 3**，line 4）．アイテムの読み出しに際し，GET*方式ではまずアドレスキャッシュにアイテムのアドレスが存在するかを調べる（**Algorithm 3**，line 11）．もし存在する場合，

Algorithm 3 GET* (Client-Side)

```
 1: payload: message payload with attributes ⟨key, value, timestamp, metadata⟩
 2: return[k]: a set of returned payload for key k
 3: buffer[k]: a set of returned payload and its address for key k
 4: addressCache[k] : remote address for key k
 5: latestTS[k] : latest timestamp for key k
 6:
 7: procedure CLIENT_GET*(K: a set of keys)
 8:     return ← {∅}
 9:     buffer ← {∅}
10:     parallel-for k ∈ K do
11:         if addressCache.contains(k) then
12:             return[k] ← RDMA-Read ⟨addressCache[k]⟩
13:             if return[k] is invalid then
14:                 RDMA-Write ⟨GET, k, ∅ ⟩
15:                 buffer[k] ← Poll response of SERVER_GET+
16:                 return[k] ← buffer[k].payload
17:                 addressCache[k] ← buffer[k].address
18:             end if
19:         else
20:             RDMA-Write ⟨GET, k, ∅ ⟩
21:             buffer[k] ← Poll response of SERVER_GET+
22:             return[k] ← buffer[k].payload
23:             addressCache[k] ← buffer[k].address
24:         end if
25:     end parallel-for
26:
27:     # Following is for detecting fractured read
28:     latestTS ← {∅}
29:     for responce r ∈ return do
30:         for k_tx ∈ r.metadata do
31:             latestTS[k_tx] ← max(latestTS[k_tx], r.timestamp)
32:         end for
33:     end for
34:     parallel-for k ∈ K do
35:         if latestTS[k] > return[k].timestamp then
36:             RDMA-Write ⟨GET, k, latestTS[k]⟩
37:             buffer[k] ← Poll response of SERVER_GET+
38:             return[k] ← buffer[k].payload
39:         end if
40:     end parallel-for
41: end procedure
```

Algorithm 4 GET* (Server-Side)

```
1: procedure SERVER_GET*(k: key, ts_req: timestamp)
2:     if ts_req = ∅ then
3:         v ← versions[k] ∧ v.timestamp = latestCommit[k]
4:         RDMA-Write ⟨v, address of v⟩
5:     else
6:         v ← versions[k] ∧ v.timestamp = ts_req
7:         RDMA-Write ⟨v, ∅⟩
8:     end if
9: end procedure
```

クライアントは RDMA-Read を実行しアイテムを読み出す（**Algorithm 3**，line 12）．存在しない場合，クライアントは RDMA-Write を用いて get リクエストを送り（**Algorithm 3**，line 14），サーバは RDMA-Write を用いてアイテムとそのアドレスを一緒に返す（**Algorithm 4**，line 3–4）．クライアントはそれらをポーリングによって取得し，アドレスキャッシュを更新する（**Algorithm 3**，line 15–17）．

サーバはクライアントによる RDMA-Read の存在を検知できないため，クライアントは RDMA-Read によって他のトランザクションが更新中のアイテムを読み出してしまう可能性がある．そこで，GET*方式ではアイテムに無効化ビットを与え，クライアント側で競合の検知を行う．もし無効化ビットが 1 になっていた場合，クライアントは RDMA-Write を用いて get リクエストを送る（**Algorithm 3**，line 13–18）．サーバはメッセージバッファをポーリングして，リクエストを取得し，処理した結果を同様に RDMA-Write でクライアントに返す．クライアントはメッセージバッファをポーリングすることで結果の取得を行う．fractured read の検知を行う部分（**Algorithm 3**，line 28–33）については RAMP トランザクションとまったく同じ内容であり，アイテムとともに格納されているメタデータを用いて検知を行う．正しいバージョンの再取得を行う部分（**Algorithm 3**，line 34–40）についても，get リクエストを RDMA-Write で送る点以外は RAMP トランザクションと同じ内容である．

4. 設計と実装

この章では，本研究で作成したプロトタイプ In-Memory Key-Value Store（KVS）の設計と実装について述べる．実装には C/C++言語を利用し，コードの行数は合計で 3,240 行となった．コンパイラには g++ (ver.4.9.3) を使用し，ライブラリは RDMA Communication Manager，MessagePack，Intel TBB，Boost ライブラリを使用した．コードは GitHub で公開している [27]．KVS は Client と Server の 2 つのプログラムから構成され，**図 3** にシステム全体の構成を示す．Client，Server はそれぞれ複数のモジュールから構成されており，共通のモジュールとして Item，Config，Communicator モジュールがある．Server は Table，Request Executor，Communicator モジュールから構成され，Client は Transaction Queue，Transaction Handler，Connection Pool モジュールから構成される．以下では，各モジュールについて解説する．

4.1 共通モジュール

Item モジュールは Key-Value データを管理する．KVS では，アイテムはすべてマルチバージョンで管理されるため，Key-Value のペアとあわせてタイムスタンプを保持する．また，メタデータとして，トランザクションによって更新される他のアイテムの一覧を Key のリストで保持する．

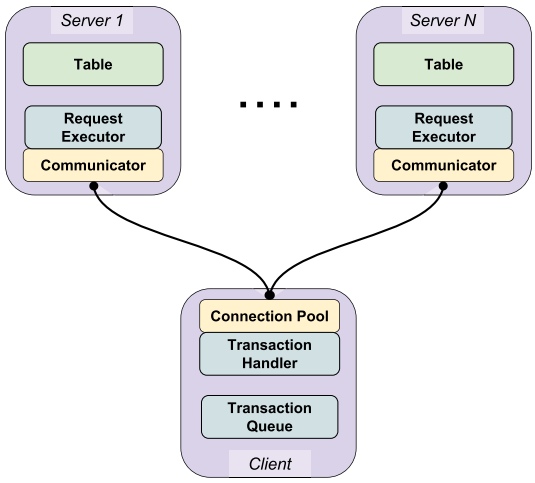

図 3　システム全体の構成
Fig. 3　Architecture.

Config モジュールはシステム全体の動作を制御する各種設定を管理する．本研究では複数の通信方式および，トランザクション処理方式を扱うため，それらの設定をコマンドライン引数として渡し，本モジュールで管理する．本モジュールは Singleton パターンを用いて実装されており，他のモジュールからグローバル変数のように参照できる．

Communicator モジュールは通信インタフェースを提供する．本研究では，TCP/IP，IPoIB，Send/Recv-Verbs，RDMA-Write/Read といった複数の通信方式を扱い，各通信方式はそれぞれ異なるインタフェースを持つ．そこで，本モジュールはそれらのインタフェースを隠蔽，抽象化し，同一のインタフェースを提供する．本モジュールでは，シリアライザとして MessagePack ライブラリを使用した．また，InfiniBand を利用する通信には RDMA Communication Manager ライブラリを使用した．Send/Recv-Verbs と RDMA-Write/Read では，あらかじめデータ転送に利用するメモリ領域を OS に登録して管理する必要があるため，バッファ管理には Boost ライブラリが提供する circular_buffer を使用している．

4.2 Client

Transaction Handler モジュールは Transaction Queue からトランザクションを取得し，実行する．Strict 2PL や RAMP トランザクションといった処理方式の設定を Config モジュールから取得し，設定に応じてモジュールの動作を切り替える．各オペレーションに対応するコネクション（Communicator）を Connection Pool から取り出し，サーバにリクエストを送る．マルチスレッド処理には Intel TBB ライブラリを使用している．

Connection Pool モジュールはクライアントとサーバ

間のコネクション（Communicator）を管理する．RDMAを用いる場合，コネクションの確立が非常に高コストであるため，本モジュールでは，初期化時に全サーバとのコネクションを確立しトランザクションを処理する間，コネクションプーリングによってコネクションを保持し続ける．本モジュールは複数スレッドから同時にアクセスされるため，スレッドセーフなコンテナとして，Intel TBBライブラリが提供するconcurrent_unorderd_map，concurrent_unordered_setを使用している．

Transaction Queue モジュールはTransaction Handlerによって処理されるトランザクションをキューに入れて管理する．現実的には，キューからトランザクションが取り出されて処理されていくと同時に新たに発行されたトランザクションがキューに追加されていくが，KVSでは初期化時にあらかじめ設定された数のトランザクションをキューに入れておき，以降新たにトランザクションを追加することはしない．したがって，キューに対する書き込みの競合は発生しない．本モジュールは複数のTransaction Handlerから同時にアクセスされるため，スレッドセーフなコンテナとして，Intel TBBライブラリが提供するconcurrent_queueを使用している．

4.3 Server

Request Executor モジュールはクライアントから送られてきたリクエストを処理する．サーバはTransaction Handlerからの接続要求を受け取ると本モジュールを起動させ，リクエストの受け付けを開始する．トランザクション処理方式の設定をConfigモジュールから取り出し，設定に応じてモジュールの動作を切り替える．Communicatorからリクエストを取り出し，処理した後，結果をCommunicatorでClientに返す．Transaction Handlerからの切断要求があるとコネクションを切断し，終了する．Strict 2PLの実装にはpthreadライブラリのread-writeロックを使用している．

Table モジュールはItemをテーブルとして管理する．Itemを管理するテーブルはタイムスタンプをkeyとし，Itemの可変長配列をvalueとするハッシュマップとなっている．テーブルは複数のRequest Executorから同時にアクセスされるため，スレッドセーフなコンテナとしてIntel TBBライブラリが提供するconcurrent_vector，concurrent_unordered_mapを使用している．

5. 評価

5.1 実験環境

実験には，自作のマイクロベンチマークとYahoo! Cloud Serving Benchmark（YCSB）[9]を用いた．YCSBはNoSQL向けのベンチマークツールとして広く利用されており，RAMPトランザクション[6]やHydraDB[23]などの研究でもYCSBによる評価が行われている．各ベンチマークではまず，ランダムに生成された文字列データをKVSに格納し，その後各種設定を変えてワークロードを実行する．なお，ワークロードは実際に外部キー制約や二次索引の管理を含む意味のある処理を実行するものではなく，単純にStrict 2PLやRAMPトランザクションといった処理方式を何度も実行するというものである．

表 1　実験環境
Table 1　Experiment environment.

OS	CentOS release 6.7 (Final)
CPU	Intel(R) Xeon(R) CPU E5620 @ 2.40 GHz x 2
コア数	2 x 4
メモリ	24 GB
Ethernet	Broadcom Corporation NetXtreme II BCM5709 Gigabit Ethernet (rev 20)
InfiniBand	Mellanox Technologies MT26428 [ConnectX VPI PCIe 2.0 5GT/s - IB QDR / 10 GigE] (rev b0)

サーバとクライアントの実行に使用したマシンのスペックを**表 1** に示す．

5.2 通信方式の比較

この実験では，RDMAを用いた通信方式の性能を，他の通信方式と比較して評価する．比較する通信方式は次のとおりである．(1) **Eth** 方式：Ethernet上でTCP/IP通信を行う方式．(2) **IPoIB** 方式：InfiniBand上でIP通信を行う方式．データ転送サービスには，Reliable Connectionを利用する．Reliable ConnectionはTCPに相当するデータ転送サービスであり，決められた相手と1対1の通信を行うものである．また，データロスの検知や再送制御を行う機能があり，信頼性のある通信が実現される（以降のInfiniBandを用いた通信方式はすべてReliable Connectionを利用するものとする）．(3) **Verbs** 方式：InfiniBand上でSend/Recv-Verbsによる通信を行う方式．(4) **RDMA** 方式：提案手法であるGET+/PUT+方式で利用される通信方式であり，InfiniBand上でRDMA-Writeによる通信を行う方式．

実験には自作のマイクロベンチマークを使用し，readの割合が非常に高い（95% read，5% write）ワークロードを実行する．トランザクション件数を100万件，トランザクションに含まれるオペレーション数を8に設定する．サーバを4プロセスで起動し，クライアントは1プロセス（8クライアントスレッド）を起動してRAMPトランザクションを実行する．

実験結果を**図 4** に示す．実験結果より，Eth方式と比較して，IPoIB方式は約1.32倍，Verbs方式は約2.39倍，

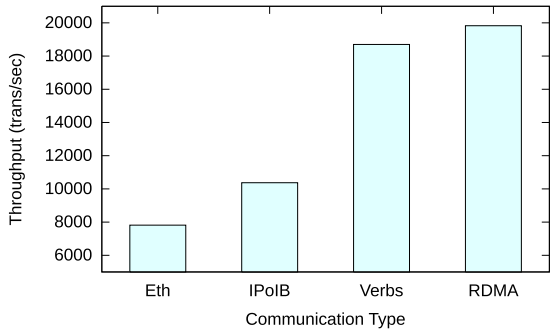

図 4　通信方式の比較（RAMP トランザクション）
Fig. 4　Comparing communication methods (RAMP transaction).

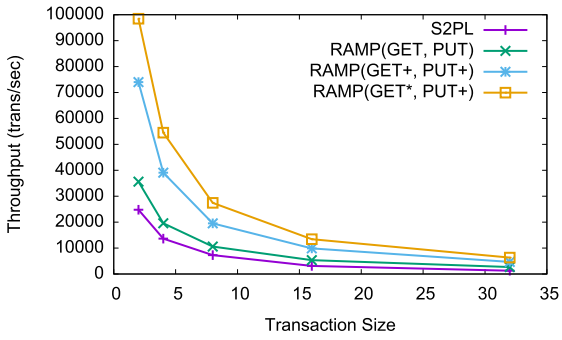

図 5　トランザクションサイズによるスループットの変化
Fig. 5　Throughput vs. transaction size.

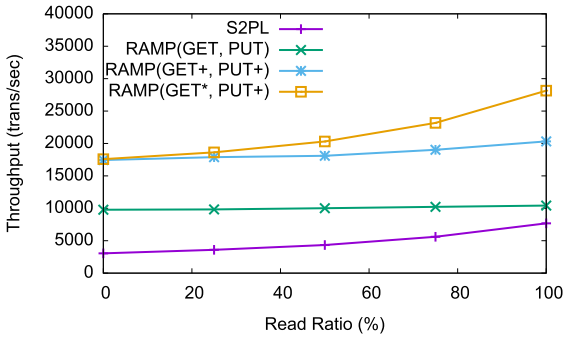

図 6　Read の割合によるスループットの変化（マイクロベンチマーク）
Fig. 6　Throughput vs. read ratio (Micro benchmark).

RDMA 方式は約 2.53 倍の性能向上を確認した[*2]．以降の実験では，従来手法の通信方式に IPoIB 方式を，提案手法の通信方式には RDMA 方式を利用し．Eth 方式と Verbs 方式の結果は省略する．

5.3　トランザクションサイズを変える実験

この実験では，トランザクションに含まれるオペレーション数（トランザクションサイズ）を変えて，各トランザクション処理方式の性能評価を行う．比較する処理方式は次のとおりである．(1) **S2PL** 方式：通信に IPoIB を利用し，Strict 2PL を行う方式．(2) **RAMP(GET, PUT)** 方式：通信に IPoIB を利用した RAMP トランザクション．(3) **RAMP(GET+, PUT+)** 方式：GET+, PUT+方式を適応した RAMP トランザクション．(4) **RAMP(GET*, PUT+)** 方式：GET*, PUT+方式を適応した RAMP トランザクション．

実験には自作のマイクロベンチマークを使用し，read の割合が非常に高い（95% read，5% write）ワークロードを実行する．トランザクション件数は 100 万件に設定する．サーバは 4 プロセスを起動し，クライアントは 1 プロセス（8 クライアントスレッド）を起動してトランザクションサイズを変えて実行する．

実験結果を**図 5** に示す．実験結果より，トランザクションサイズが 4 のとき，RAMP(GET, PUT) 方式と比較して，RAMP(GET+, PUT+) 方式は約 2.07 倍，RAMP(GET*, PUT+) は約 2.78 倍の性能向上を確認した．また，RAMP(GET+, PUT+) 方式と比較して RAMP(GET*, PUT+) 方式は約 1.39 倍の性能向上を確認し，RAMP(GET*, PUT+) 方式が最も高い性能を示すことを確認した．また，オペレーションの数が増えるに従ってスループットの低下を確認した．これは，トランザクションに含まれるオペレーションの数が増えるに従って，1 トランザクションを処理する時間が増加するためである．

[*2] 本実験結果における RAMP トランザクションの性能は，文献 [6] における RAMP トランザクションの性能に比べて 5 倍程度低いものとなっている．これは，文献 [6] における実験と比べて本実験はクライアント数，サーバ数ともに小規模であり，クライアントのコネクション数が少ないことが原因の 1 つだと考えられる．また，文献 [6] では multi-versioning や garbage collection などの最適化が行われているが，本研究では，それらを実装できていないことも原因の 1 つだと考えられる．

5.4　Read の割合を変える実験（マイクロベンチマーク）

これまでの実験では，同じ read の割合（95% read，5% write）でワークロードを実行させていた．そこで，この実験ではワークロードにおける read の割合を変化させて各トランザクション処理方式の性能にどのような影響を与えるのかを調査する．

実験には自作のマイクロベンチマークを利用し，トランザクション件数を 100 万件，トランザクションサイズを 8 に設定する．サーバは 4 プロセスを起動し，クライアントは 1 プロセス（8 クライアントスレッド）を起動して read の割合を変化させて実行する．

実験結果を**図 6** に示す．実験結果より，read の割合が 0%の場合，RAMP(GET*, PUT+) 方式は RAMP(GET+, PUT+) 方式と同じ性能を示すことを確認した．また，read の割合が高くなるに従って，RAMP(GET*, PUT+) 方式が RAMP(GET+, PUT+) 方式よりも高い性能を示すことを確認した．read の割合が 100%の場合，RAMP(GET, PUT) 方式と比較して，RAMP(GET+, PUT+) 方式は約 1.94 倍，RAMP(GET*, PUT+) 方式は約 2.69 倍の性能向上を確認した．また，RAMP(GET+, PUT+) 方式と比較

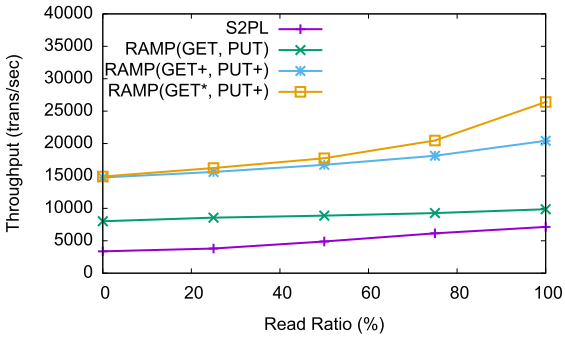

図 7　Read の割合によるスループットの変化（YCSB）
Fig. 7　Throughput vs. read ratio (YCSB).

して RAMP(GET*, PUT+) 方式は約 1.38 倍の性能向上を確認した．

5.5　Read の割合を変える実験（YCSB）

この実験では，ベンチマークに YCSB を利用し，read の割合を変化させて各トランザクション処理方式の性能評価を行う．YCSB は Java によって実装されているベンチマークツールであるため，そのままだと本研究で作成した C++による KVS と組み合わせることができない．そこで，Java Native Interface（JNI）を利用し，YCSB から KVS の関数を呼び出す．また，YCSB にはトランザクションの機能が存在せず，複数のオペレーションを 1 つの単位として実行できない．そこで，YCSB のコードを修正し，設定した数のオペレーションをまとめて実行する機能を新たに追加した．YCSB のコードを修正するにあたり，Bailis が公開している RAMP トランザクションのコード [28] を参考に，同様の修正を行った．まず，YCSB の設定パラメータにトランザクションサイズの項目を追加し，コマンドライン引数もしくは設定ファイルからトランザクションサイズを設定できるようにした．さらに，ワークロードを実行する部分を修正し，YCSB が get/put オペレーションを 1 つ発行すると，内部的にトランザクションサイズ分のオペレーションが発行されるようにした．

YCSB には，あらかじめ 6 つの基本的なワークロード：Workload A～F が用意されており，それらは設定ファイル workloada～workloadf を読み込むことで利用できる．この実験では，これらの設定ファイルをコピーして read の割合を 0%，25%，50%，75%，100%に変更した設定ファイル：workload_r0～workload_r100 を新たに作成して用いる．各設定ファイルに共通する設定として，データサイズを 1 KB，テーブルサイズを 1000，データ分散方式を uniform に設定する．サーバは 4 プロセスを起動し，クライアントは 1 プロセス（8 クライアントスレッド）を起動する．

実験結果を図 7 に示す．実験結果は，マイクロベンチマークによる実験結果と同様の傾向を示し，read の割合が 100%の場合，RAMP(GET, PUT) 方式と比較して，RAMP(GET+, PUT+) 方式は約 2.06 倍，RAMP(GET*, PUT+) 方式は約 2.67 倍の性能向上を確認した．また，RAMP(GET+, PUT+) 方式と比較して RAMP(GET*, PUT+) 方式は約 1.29 倍の性能向上を確認した．

6.　結論

本研究では，高性能インターコネクトである InfiniBand を利用し，RDMA の機能を用いて RAMP トランザクションを高速化する手法を提案した．まず，RDMA-Write による GET/PUT オペレーションの高速化手法として GET+/PUT+方式を提案し，次に，RDMA-Read による GET+方式のさらなる高速化手法として GET*方式を提案した．提案手法の評価のため，C/C++言語を用いてプロトタイプ In-Memory Key-Value Store を実装し，各通信方式および，各トランザクション処理方式の評価を行った．YCSB による実験結果より，通信に IPoIB を利用する RAMP トランザクションと比較して GET+/PUT+方式を適応することで最大 2.06 倍の高速化を達成した．さらに，GET*方式を適応することで最大 2.67 倍の高速化を達成した．この結果から RDMA-Write と RDMA-Read を効果的に用いることで，RAMP トランザクションを高速化できることが分かった．我々は提案方式ならびにコード [27] が InfiniBand を用いる分散 KVS に有益だと信じる．

謝辞　本研究の一部は，JST CREST「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア」，JST CREST「EBD：次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」，JST CREST「広域撮像探査観測のビッグデータ分析による統計計算宇宙物理学」の支援を受けたものである．

参考文献

[1] Gant, J. and Reinsel, D.: IDC Report, The digital universe in 2020: Big data, bigger digital shadows and biggest growth in the Far East (2012).
[2] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data, *ACM Trans. Comput. Syst.*, Vol.26, No.2 (2008).
[3] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store, *SOSP*, pp.205–220 (2007).
[4] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H.C., Marchukov, M., Petrov, D., Puzar, L., Song, Y.J. and Venkataramani, V.: TAO: Facebook's Distributed Data Store for the Social Graph, *USENIX*, pp.49–60 (2013).
[5] Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H., Puz, N., Weaver, D. and Yerneni, R.: PNUTS: Yahoo!'s Hosted Data Serving Platform, *PVLDB*, Vol.1, No.2, pp.1277–1288 (2008).

[6] Bailis, P., Fekete, A., Hellerstein, J.M., Ghodsi, A. and Stoica, I.: Scalable atomic visibility with RAMP transactions, *SIGMOD*, pp.27–38 (2014).
[7] InfiniBand Trade Association: InfiniBand Architecture Specification, available from ⟨http://www.infinibandta.org/⟩.
[8] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S. and Paleczny, M.: Workload Analysis of a Large-scale Key-Value Store, *SIGMETRICS*, pp.53–64 (2012).
[9] Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R. and Sears, R.: Benchmarking cloud serving systems with YCSB, *SoCC*, pp.143–154 (2010).
[10] Oracle: Oracle Database, available from ⟨http://www.oracle.com/database/overview/index.html⟩.
[11] The PostgreSQL Global Development Group: PostgreSQL, available from ⟨http://www.postgresql.org/⟩.
[12] Oracle: MySQL, available from ⟨https://www.mysql.com/⟩.
[13] Basho: Riak, available from ⟨http://basho.com/products/riak-kv/⟩.
[14] Gray, J. and Lamport, L.: Consensus on Transaction Commit, *ACM Trans. Database Syst.*, Vol.31, No.1, pp.133–160 (2006).
[15] TOP500.org: List Statistics, available from ⟨http://www.top500.org/statistics/list/⟩.
[16] Mitchell, C., Geng, Y. and Li, J.: Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store, *USENIX ATC*, pp.103–114 (2013).
[17] さくらインターネット株式会社：さくらインターネット，演算に特化した「高火力コンピューティング」への取り組みを開始～Infiniband 接続による大規模な GPU クラスタを Preferred Networks 社と共同構築～，入手先 ⟨https://www.sakura.ad.jp/press/2016/0126_gpu/⟩ (アクセス日：2016/05/09).
[18] Mellanox Technologies：ヤフー株式会社，メラノックス InfiniBand 製品を採用，入手先 ⟨http://www.mellanox.co.jp/news/press20140930_MLNX_Yahoo_Japan.html⟩ (アクセス日：2016/05/09).
[19] Microsoft Japan: Windows Azure クラウドサービスに InfiniBand 採用の A8，A9 インスタンスを追加，入手先 ⟨https://blogs.msdn.microsoft.com/bluesky/2014/01/30/windows-azure-infiniband-a8a9/⟩ (アクセス日：2016/05/09).
[20] Klaff, B.: Introduction to InfiniBand, available from ⟨http://www.mellanox.com/blog/2014/09/introduction-to-infiniband⟩ (アクセス日：2016/05/09).
[21] Daikoku, H., Kawashima, H. and Tatebe, O.: On Exploring Efficient Shuffle Design for In-Memory MapReduce, *Proc. 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond* (2016).
[22] Kalia, A., Kaminsky, M. and Andersen, D.G.: Using RDMA Efficiently for Key-value Services, *SIGCOMM*, pp.295–306 (2014).
[23] Wang, Y., Zhang, L., Tan, J., Li, M., Gao, Y., Guerin, X., Meng, X. and Meng, S.: HydraDB: A Resilient RDMA-driven Key-Value Middleware for In-memory Cluster Computing, *SC*, pp.22:1–22:11 (2015).
[24] Dragojevic, A., Narayanan, D., Nightingale, E.B., Renzelmann, M., Shamis, A., Badam, A. and Castro, M.: No compromises: Distributed transactions with consistency, availability, and performance, *SOSP*, pp.54–70 (2015).
[25] Frey, P.W., Goncalves, R., Kersten, M.L. and Teubner, J.: Spinning relations: High-speed networks for distributed join processing, *DaMoN*, pp.27–33 (2009).
[26] 三橋龍也，川島英之，建部修見：環結合による類似検索の高性能化，情報処理学会第 150 回 HPC 研究会報告 (HPC150)，Vol.2015-HPC-150, No.16 (2015).
[27] Murata, N.: Source Code of RAMP with RDMA, available from ⟨https://github.com/nao23/ramp-with-rdma⟩.
[28] Bailis, P.: Code for "Scalable Atomic Visibility with RAMP Transactions" in SIGMOD 2014, available from ⟨https://github.com/pbailis/ramp-sigmod2014-code⟩ (アクセス日：2016/05/09).

村田 直郁

1994 年生．2016 年筑波大学情報学群情報科学類卒業．同年同大学大学院システム情報工学研究科コンピュータサイエンス専攻入学．

川島 英之　（正会員）

1999 年慶應義塾大学理工学部電気工学科卒業．2005 年同大学大学院理工学研究科開放環境科学専攻後期博士課程修了．同年慶應義塾大学理工学部助手．2007 年筑波大学大学院システム情報工学研究科講師，ならびに計算科学研究センター講師．2016 年筑波大学計算科学研究センター准教授．博士（工学）．データ基盤に興味を持つ．日本データベース学会，ACM，IEEE 各会員．

建部 修見　（正会員）

1969 年生．1992 年東京大学理学部情報科学科卒業．1997 年同大学大学院理学系研究科情報科学専攻博士課程修了．同年電子技術総合研究所入所．2005 年独立行政法人産業技術総合研究所主任研究員．2006 年筑波大学大学院システム情報工学研究科准教授．2015 年同教授．博士（理学）（東京大学）．超高速計算システム，グリッドコンピューティング，並列分散システムソフトウェアの研究に従事．日本応用数理学会，ACM 各会員．

（担当編集委員　小林　大）