Exploiting Distributed Version Concurrency in a Transactional Memory Cluster

Kaloian Manassiev
Department of Computer Science, University of Toronto, Canada
kaloianm@cs.toronto.edu

Madalin Mihailescu
Department of Computer Science, University of Toronto, Canada
madalin@cs.toronto.edu

Cristiana Amza
Department of Electrical and Computer Engineering, University of Toronto, Canada
amza@eecg.toronto.edu

Abstract

We investigate a transactional memory runtime system providing scaling and strong consistency for generic C++ and SQL applications on commodity clusters. We introduce a novel page-level distributed concurrency control algorithm, called Distributed Multiversioning (DMV). DMV automatically detects and resolves conflicts caused by data races for distributed transactions accessing shared in-memory data structures. DMV's key novelty is in exploiting the distributed data versions that naturally occur in a replicated cluster in order to avoid read-write conflicts. Specifically, DMV runs conflicting transactions in parallel on different replicas, instead of using different physical data copies within a single node as in classic multiversioning.

In its most general update-anywhere configuration, DMV can be used to implement a software transactional memory abstraction for classic distributed shared memory applications. DMV supports scaling for highly multithreaded database applications as well by centralizing updates on a master replica and creating the required page versions for read-only transactions on a set of slaves. In this DMV configuration, a version-aware scheduling technique distributes the read-only transactions across the slaves in such a way to minimize version conflicts.

In our evaluation, we use DMV as a lightweight approach to scaling a hash table microbenchmark workload and the industry-standard e-commerce workload of the TPC-W benchmark on a commodity cluster. Our measurements show scaling for both benchmarks. In particular, we show near-linear scaling up to 8 transactional nodes for the most common e-commerce workload, the TPC-W shopping mix. We further show that our scaling for the TPC-W e-commerce benchmark compares favorably with that of an existing coarse-grained asynchronous replication technique.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming—Distributed programming

General Terms Measurement, performance, experimentation

Keywords Transactions, in-memory, scalability, concurrency control, replicated databases

PPoPP'06 March 29–31, 2006, New York, New York, USA.

1. Introduction

In this paper we introduce and evaluate a novel page-level distributed concurrency control algorithm, called Distributed Multiversioning (DMV). DMV allows transactions on different machines to manipulate shared in-memory data structures in an atomic and serializable manner. DMV forms the basis of a software transactional memory system for scaling both C++ and SQL applications on a commodity cluster.

Software transactional memory (STM) [14, 16, 17, 18] has recently emerged as a novel parallel programming paradigm for facilitating efficient, programmer-friendly use of the plentiful parallelism available in hardware. Instead of synchronization, the user inserts delimiters for transactions to be executed in parallel by the STM run-time system. The STM automatically detects and resolves conflicts caused by data races. Typically, software transactional memory is implemented as a multiversioned in-memory object store inside a single multiprocessor node. Each new transaction creates a new copy of each object, either upon first access [18], or upon the first update to that object. Similar copy-on-write techniques have also been traditionally used in multiversioned databases (e.g., PostgreSQL). Such systems pay the price of maintaining multiple physical data item copies and of garbage collecting old copies.

We observe that, in a distributed system, maintaining several distributed versions of the same data item, one at each distributed node, comes almost for free. In the common case, an update can upgrade an item replica at a particular node to a new version, while a read proceeds concurrently on the old version of the same data item at another node. Since conflicts occur for a limited duration, a replicated cluster system may thus provide a sufficient number of readily available data item versions in order to avoid most conflict waits. Based on this observation, Distributed Multiversioning (DMV) combines fine-grained (per-page) concurrency control with asynchronous data replication on a cluster in order to improve read-write concurrency. DMV provides scaling and strong consistency (i.e., 1-copy serializability [9]) for the overall cluster system, while maintaining a single copy of the application data at each node.

In DMV, each update transaction executing at a node creates a new version of the shared memory. The update transaction broadcasts per-page version-tagged modifications to all other replicas as a pre-commit action. Each replica receiving an update broadcast delays applying the modifications locally and replies immediately, thus not delaying the committing update transaction. If the modification broadcast is received during an update transaction and a conflict is detected, the local transaction is rolled-back and restarted.

The key optimization in DMV is to isolate the execution of a read-only transaction from any remote conflicting updates. A read-only transaction creates its own consistent "snapshot" [9] on the

local replica by *selectively* applying outstanding modifications on the shared pages it touches. Creating per-page versions lazily allows different read-only transactions with *disjoint* read sets to run concurrently at the same replica even if they require different versions for their items. Conversely, if two read-only transactions need two different versions of the same item, they can only execute in parallel if sent to different replicas. We leverage the presence of a scheduler that distributes transactions on a cluster in many application configurations in order to avoid version conflicts. Specifically, a version-aware scheduler sends transactions requiring conflicting versions to different cluster nodes. In the case of imperfect scheduling (e.g., due to insufficient replicas), a read-only transaction may need to wait for other co-located transactions using a previous version of an item to finish in order to create its own. Moreover, the price we pay for not keeping older versions around is that a read-only transaction may be aborted if the version it needs for a particular page has been already overwritten by another transaction carrying a higher version number. Fortunately, this case is rare in practice.

In this paper we focus on two DMV cluster configurations: update-anywhere with no scheduler support and master-update with scheduler support. The update-anywhere DMV configuration supports scaling through STM techniques for classic C++ distributed shared memory applications [4]. In the master-update DMV configuration, a scheduler minimizes conflicts by sending all update transactions on a master replica and distributing read-only transactions across a set of slave replicas in a version-aware manner. The master-update DMV configuration provides scaling for highly multithreaded applications, such as, database applications. For these applications, the probability of read-write conflicts occuring within a node is high in the update-anywhere configuration.

In order to aid our implementation efforts, our replicated transactional memory system borrows heavily from an existing distributed shared memory library: the TreadMarks [4] library, which provides automatic detection and encapsulation of per-page modifications. In addition to TreadMarks, in order to support SQL-based transactional applications, we also build on the MySQL "heap-table" code [1]. This version of MySQL provides a very simple and efficient in-memory SQL database engine *without transactional properties* using red-black trees [12] as index data-structure.

In our evaluation, we use the three workload mixes of the industry standard TPC-W e-commerce benchmark [3] as our SQL-based benchmark and a distributed hash table microbenchmark as an example of support for scaling generic C++ applications. For the distributed hash table microbenchmark, we explore a range of workload mixes by varying the size and frequency of the hash table insert/delete transactions versus hash table lookup transactions. For TPC-W, we use the three standard workload mixes of TPC-W, which similarly allow us to vary the fraction of write-type transactions from 5% to 50%: browsing (5%), shopping (20%) and ordering (50%). We have implemented the TPC-W web site using popular open source software packages: the Apache Web server [8], the PHP Web-scripting/application development language [23], and the MySQL heap-table database server. We have modified the MySQL heap-table code to include transactional calls to our runtime system, thus obtaining a lightweight in-memory tier. We use this tier in place of the database back-end for the TPC-W e-commerce site.

We further compare the throughput scaling obtained through Distributed Multiversioning on an e-commerce database cluster of one master and eight slave replicas with the throughput and scalability of a state-of-the-art replication technique for e-commerce applications, Conflict-Aware scheduling [5]. This technique is representative of recently proposed asynchronous database replication schemes offering strong consistency guarantees [19, 11]. Similar to DMV, this scheme distributes read-only requests on a cluster of database replicas and replicates writes lazily for scaling. On the other hand, the conflict resolution employed by this previous scheme is coarser grained, per-table instead of per-page as in DMV.

Our results are as follows:

1. Distributed Multiversioning provides close to linear scaling for the browsing and shopping TPC-W workloads and limited scaling for the write-heavy ordering mix of TPC-W.
2. The scaling of DMV compares favorably with the throughput scaling of the asynchronous replication technique with coarse-grained conflict resolution.
3. Our in-memory transactional system also provides support for scaling the hash table microbenchmark in the classic distributed shared memory configuration.

The rest of this paper is organized as follows. Section 2 introduces the necessary background in consistency maintenance techniques used in software distributed shared memory. Section 3 introduces our Distributed Multiversioning scaling solution. Section 4 gives details on our prototype implementation. Sections 5 and 6 describe our experimental environment and results. Section 7 discusses related work. Section 8 provides our conclusions.

2. Background

In this section we provide the necessary background in software distributed shared memory (SDSM). Specifically, we describe the basics of a state-of-the-art SDSM, TreadMarks [4]. TreadMarks is a user-level SDSM system that runs on commodity clusters. TreadMarks provides parallel programming primitives similar to those used in hardware shared memory machines, namely, process creation, shared memory allocation, and lock and barrier synchronization.

TreadMarks relies on user-level memory management techniques provided by the operating system to detect accesses to shared memory at the granularity of a page. A multiple-writer protocol is employed to reduce the amount of communication involved in implementing the shared memory abstraction. Two or more nodes can simultaneously modify their own copy of a shared page. The modification merge is accomplished through the use of *diffs*. A diff is a runlength encoding of the modifications made to a page, generated by comparing the page to a copy, called *twin* saved prior to the modifications. Even though DMV uses a different protocol for consistency maintenance and introduces alternative access trapping mechanisms compared to TreadMarks, we reuse the basic mechanisms for process creation, shared memory allocation, encapsulating and merging modifications by twinning and diffing.

3. Distributed Multiversioning

In this section, we introduce our Distributed Multiversioning (DMV) protocol that allows application threads to manipulate shared in-memory data structures in a distributed manner through a transactional API. The goal of DMV is to scale the application on a cluster through a novel distributed concurrency control mechanism that integrates fine-grained concurrency control and strongly consistent lazy replication.

3.1 Application Programming Interface

This section describes the transaction API and the related programming paradigms that our runtime system offers. All applications access shared memory through transactions. There is no attempt to support non-transactional execution. Furthermore, each application transaction executes only at a single machine. Multi-machine transactions and distributed commit are not supported.

In case an on-disk database is used by the transactional application, the application running at each node maps the database into its virtual address space. This part of the application's address space is shared with other applications that also have the database mapped. If two applications execute on different machines, then that sharing is brought about by DMV.

There is no need for the application to do explicit locking. Appropriate concurrency control to guarantee serializable execution is done by DMV on behalf of the application. DMV may also abort a transaction in order to maintain serializability.

Our system offers the following set of primitives as its basic API.

```
init_transactions()
begin_transaction()
commit_transaction()
abort_transaction()
allocate_dtmemory()
```

`Begin transaction`, `commit transaction`, and `abort transaction` implement the customary transaction semantics [15]. The `init transactions` operation performs system initializations and should be called before the start of the first transaction in the application code. The `allocate dtmemory` operation allocates shared memory space of a required size in the transactional memory. The memory allocation scheme is similar to the one in TreadMarks [4]. In addition, as in TreadMarks [4], primitives for remote process creation are also supported.

More importantly, because an application may access both local and shared data, the application needs to explicitly declare variables that are part of the shared transactional memory.

3.2 Overview of DMV

Each node in our in-memory transactional cluster maintains a single physical copy of the application data at any given time. Instead of local physical copies, DMV takes advantage of the distributed updates that occur anyway in a cluster system for consistency maintenance, hence of the availability of distributed data item replicas. If a single application thread executes at each cluster node, then an update and a read-only transaction always occur on different replicas, hence they do not interfere with each other. Our key idea for supporting multithreaded execution within each node is to allow each read-only transaction to create the consistent "snapshot" that it needs, dynamically, on-demand, at a particular replica for the pages in its read set. In this way, several concurrent read-only transactions that require different snapshots, but have disjoint read-sets can each create their respective snapshots with the required versions on the same physical copy of the transactional memory.

In the following, we present two DMV configurations. The first assumes a typical STM application environment where, instead of synchronization, the user inserts delimiters for transactions to be executed speculatively in parallel on a cluster by the run-time system. Next, we present a special case that is important in practice where transactions are presented to the system through a scheduler which distributes them on a cluster of in-memory replicas. The scheduler is leveraged to distribute transactions across the cluster in such a way that conflict waits and aborts are optimized. In particular, the scheduler-based approach is useful for applications where several threads are expected to run at each node. In this case, in the absence of multiversioning within the node, read-only transactions would normally block conflicting update transactions resulting in potentially long conflict waits. The scheduler avoids these conflict waits by scheduling all update transactions on a master node and read-only transactions on a set of slave replicas. Furthermore, the scheduler distributes the read-only requests across the cluster through a version-aware scheme in order to avoid the case where two read-only transactions need two different versions of the same data item. In the following we first present the update-anywhere DMV protocol and then the scheduler-based master-update DMV protocol.

3.3 Update-Anywhere DMV Protocol

In this section, we describe the DMV protocol in the fully decentralized cluster configuration. DMV uses an update-anywhere replication protocol that automatically detects and resolves distributed conflicts caused by data races.

3.3.1 Consistency Maintenance in Update-Anywhere DMV

During the execution of a transaction, DMV traps each read and write access. As in Treadmarks, modifications to shared memory cause twins to be created for the particular pages. Updates are only visible to other transactions upon commit. At commit, the update transaction i) creates a new (cluster-wide) version for the transactional memory, ii) broadcasts the modifications (diffs) performed during the transaction to all other nodes as a pre-commit action and iii) waits for their acknowledgments before committing the transaction locally. In order to enforce a consistent serialization order of update transactions, each update transaction obtains a unique system-wide token during commit. Only a single updater can thus perform a modification broadcast (diff flush) at any given time. The diff flush is tagged with the unique version number of the distributed transactional memory created by the committing node. All nodes receiving a diff flush store the diffs locally, but delay applying them to the corresponding pages. Instead, they update their locally maintained version number of the transactional memory and reply immediately in order to minimize the delay for the committing update transaction. The application of modifications occurs lazily only when a local transaction needs those modifications at that node. Each transaction applies the modifications stored locally to each page, on-demand, upon each page access. A page is recognized to be stale if its version number is lower than the locally maintained version number of the last commit seen by the node.

3.3.2 Conflict Resolution in Update-Anywhere DMV

The protocol differentiates between actions taken by update and read-only transactions in order to avoid aborts of read-only transactions in the case of a concurrent remote writer. For an *update transaction* if, upon a page access trap, the page is determined to be stale, *all* diffs stored locally for that page are applied to it in increasing order of version number. Conflicts between two remote *update transactions* are detected when an incoming diff flush for a page accessed locally is received while an *update transaction* is executing at the local node. In this case, if any of the pages included in the diff flush were either written or read by the local node, then the local update transaction is aborted and restarted.

On the other hand, for *read-only transactions*, the transaction creates a "snapshot" of the data that it touches consistent with the transactional memory version at the beginning of the respective transaction. The snapshot is created lazily. Upon each (read) access trap on a page, the transaction applies only the locally stored diffs with version numbers up to and including the version of the transactional store at `begin_transaction`. A read-only transaction is not affected by incoming diff flushes during its execution, hence is ordered before any concurrent update transactions in the global serialization order.

Our protocol *does not require* that transactions are manually tagged with their type. All transactions are initially classified optimistically as read-only by the run-time system. A transaction is reclassified as update upon trapping the first write access on a page. Reclassification implies a validation phase that determines whether the transaction can safely continue. If the current transaction has al-

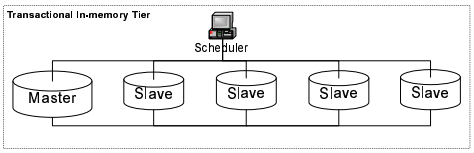


Figure 1. System design for Master-update configuration.

ready ignored a diff flush when reading a page by not applying all diffs, the transaction is rolled-back and restarted as an update transaction. Otherwise, the transaction can safely continue as an update transaction.

3.4 Scheduler-based Master-Update DMV Protocol

In this section, we describe an important special case of the DMV protocol that supports conflict wait avoidance for highly multithreaded applications, such as traditional database applications. We take advantage of the presence of a scheduler that distributes transactions on a cluster in many such application configurations. In this scheme, the scheduler is aware of the type of transactions and the versions that they are supposed to read. The scheduler uses this knowledge to schedule the execution of update transactions on a designated master replica, while distributing conflicting read-only transactions across a set of slave replicas (see Figure 1).

In this scheduler-based scheme, we currently tag each transaction with its type, in order for the scheduler to recognize opportunities for scheduling read-only transactions separately from update transactions. However, this is a choice we made in order to simplify implementation and is not mandatory. Indeed, a transaction that is optimistically assumed to be read-only can be either migrated to or restarted on the master node once we detect a first update for that transaction.

3.4.1 Consistency Maintenance in Master-Update DMV

For each update transaction, the master broadcasts modifications to the set of slaves as a pre-commit action. Obtaining a cluster-wide token is no longer necessary prior to sending the diff-flush; The master's internal concurrency control decides the global serialization order of updates. As before, each slave replica delays the application of modifications, thus not delaying the committing master database in order to provide scaling. In this way, updates on the master are not delayed by conflicting reads that may be occurring on the slaves. Each update on the master node creates a version number communicated to the scheduler that distributes requests on the in-memory cluster. The scheduler tags each read-only transaction with the newest version received from the master and sends it to one of the slaves. As before, the appropriate version for each individual data item is then created dynamically and lazily at that slave replica, when needed by an in-progress read-only transaction.

3.4.2 Conflict Resolution in Master-Update DMV

Since all updates occur on the master, there are no conflicts between distributed update transactions. The system automatically detects data races created by co-located read-only transactions attempting to read conflicting versions of the same item. Version conflicts are detected and enforced at the page level. A read-only transaction may need to wait for a previously scheduled transaction to release a lower version of a page in order to create its own. It is also possible, although less probable, that a read-only transaction T1 needs to be aborted if another read-only transaction T2 upgrades a shared item to a version higher than that already read by T1.

In the next section we introduce version-aware scheduling techniques designed to minimize the probability of conflict waits and aborts for read-only transactions.

3.4.3 Version-Aware Scheduling for Read-Only Transactions

The scheduler keeps track of the data versions that exist or are about to be created at each replica and sends a read-only transaction to execute on a replica where the chance of version conflicts is the smallest. Our current heuristic selects a replica where read-only transactions with the same version number as the one to be scheduled are currently executing, if such replicas exist. If not, the scheduler attempts to progressively relax the requirement. It proceeds to search for a replica where the number of conflicting versions for concurrently created snapshots would be 2, then 3. Otherwise it selects any replica by plain load balancing. The scheduler load balances across the candidate replicas thus selected using a shortest execution length estimate, as in the SELF algorithm we introduced in our previous work [7].

In the common case, the scheduler sends any two read-only transactions requiring different versions of the same memory page on different replicas, where each creates the page versions it needs and the two transactions execute in parallel. Since we do not assume that the scheduler has any knowledge about a transaction's working set and the replica set is finite, the occasional read-only transaction may still need to wait or can be aborted due to other read-only transactions using a previous or higher version of an item, respectively. However, we expect these situations to be rare and decrease with the number of replicas in the system.

4. Prototype Implementation Details

This section presents the implementation of access trapping and data structures in our distributed transactional runtime system.

4.1 Access Trapping and Consistency Granularity

In order to manage data races transparently to the application, we trap both read and write accesses to shared transactional memory. Access trapping for write accesses is performed through memory protection violations, as in TreadMarks. However, our system allows read accesses on pages with outstanding modifications (in read-only transactions), hence the distinction between a valid and invalid page state is not as clear-cut as in original TreadMarks. This problem is exacerbated by DMV support for multithreading within a node.

We introduce an additional access trapping mechanism based on the operator overloading mechanism in C++ that helps us trap all read accesses at lower overhead. For example, we use operator overloading for all pointer indirections on a tree node pointer type. For this purpose, the tree node data type needs to be declared as a new data type with overloaded operators. A similar operator overloading technique can be used to trap write accesses as well. In this case, assignment overloading of all assignment operators e.g., =, +=, -=, etc and increment/decrement overloading (i.e., for ++, –) is used. While fully analyzing the trade-offs between the two trapping methods is beyond the scope of this paper, we mention the following advantages and disadvantages of the operator overloading technique. Compared to memory protection violations, operator overloading is more lightweight and allows us to trap accesses at the level of a word. However, in this paper we choose a page-level consistency unit for all access trapping because it allows for some degree of aggregation when performing twinning and diffing and reduces metadata space overhead. More importantly, memory violation trapping is fully transparent. In contrast, in order to allow for operator overloading, the application writer needs to redefine all data types for transactional memory variables, including basic types, e.g., for integer variable declarations.

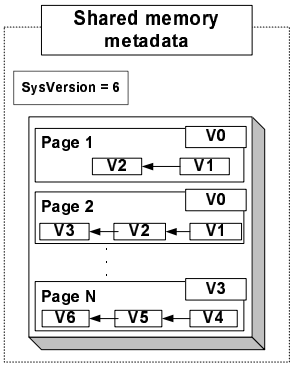


Figure 2. Per-page versioning.

4.1.1 Metadata Organization and Usage

Meta-data maintained by our run-time system for each shared memory page includes the fields depicted in Figure 2: a per-page `VersionID` illustrated in the top right corner of each page entry metadata and an outstanding `DiffQueue` shown as a list of diffs, each tagged with its corresponding version number. The `VersionID` designates the cluster-wide system version that the page currently corresponds to. The `DiffQueue` is a linked list with physical modifications to the page data (diffs), which correspond to the evolution of the page, but have not yet been applied. The `DiffQueue` is maintained in sorted version order and is later applied on-demand in increasing order of version numbers. Upon the reception of a diff-flush, a replica pre-processes the message, classifies the in-coming diffs by page ID and stores the diffs locally in the `DiffQueues` for the corresponding pages. No modifications are applied to the pages at this point. The replica node then immediately reports to the sending node that it has successfully received and stored the diffs.

Other data structures are directly inherited from the original SDSM system, Treadmarks. Specifically, the set of *twins* created during the transaction execution forms an *undo log* that is used to roll-back a transaction. *Diffs* are created at commit and are aggregated in the update modification broadcast, i.e., the diff-flush message.

4.1.2 Metadata Space Management

As new versions for pages are dynamically created, older versions are overwritten and diffs are discarded at each replica when necessary for the purposes of an on-going transaction. Any update transaction discards all diffs it has itself created upon commit. Twins are also discarded after every commit. To handle the unlikely case of a data item being continuously written by one node, but never read at a particular replica, we use a periodic examiner thread. Upon activation, the thread inspects the list of active transactions at the local replica and finds the one requiring the lowest version v_{min}. Then, it visits all shared memory pages at that replica and, for each page with version less than v_{min}, it applies all modifications up to and including v_{min}.

4.1.3 Support for Multithreaded Execution

If multiple threads execute at a node, each node resolves local conflicts by using a variant of a reader-writer two-phase-locking [9] protocol. Locks are implemented as condition variables in such a way to block threads that are waiting for a lock held in conflicting mode and to notify waiting threads when the lock is free. Furthermore, since read-only transactions create their required versions on demand, hence imply updates to pages, we modify the readers lock to allow multiple read-only transactional threads to access a shared page only if their required version is the same.

Specifically, when a read-only transaction needs to access a page, it first checks whether the version of the page corresponds to the version that the transaction expects. If that is the case, a reference count field of the page data structure is incremented. Otherwise, the transaction applies the corresponding diffs to upgrade the page to the required version first. When the transaction commits, it releases all held locks by decrementing the reference count of all pages that it accessed. If for any of these pages the reference count reaches zero, the corresponding waiting transactions are notified to proceed and retry.

In addition to lock waits, as mentioned before, a read-only transaction can be aborted due to reading a page with an inconsistent (higher) version than that of its snapshot. This case may occur due to interference with either another read-only transaction or an update transaction executing at the same node. Next, we illustrate our protocol for various cases using example scenarios.

4.1.4 Examples

Figure 3 illustrates the creation of two different simultaneous "snapshots" of the transactional memory for two different concurrently executing read-only transactions (T_1 and T_2) without keeping multiple physical copies of each data page. We assume a master-update configuration where multiple threads execute at each node. Transaction T_1 was tagged with version v_1 by the scheduler, while T_2 was assigned version v_3. Transaction T_1 reads pages P_1 and P_2 and upgrades them to version v_1, while transaction T_2 upgrades page P_N to its required version v_3 in parallel.

Figures 4 and 5 show scenarios where the two transactions have overlapping read-sets and different assigned versions. Since the order in which T_1 and T_2 arrive at the scheduler is likely to be the order of their assigned versions, normally T_1 creates and locks version v_1 on P_1 and P_2 as it touches the respective pages before T_2 even starts to execute. Subsequently, T_2 may need to wait until T_1 commits and releases its lock on page P_2 in order to be able to create its required version v_3 on P_2. This wait is due to a read-write conflict on page P_2 upon creating version v_3. Note, however, that this conflict's duration would normally be longer if the full corresponding transactions producing v_1 and v_3 at the master would have executed on the respective slave replica. Instead, our selective per-page modification application process minimizes the conflict wait.

Finally, it is possible that T_2 accesses P_2 ahead of T_1 even if arriving later at the respective replica as in Figure 5. Once T_2 commits, version v_1 of page P_2 that T_1 needs is lost. If T_1's subsequent access on P_2 is its first page access, we upgrade T_1's required version to v_3. Otherwise, as in the case shown, if T_1's access to P_2 occurs after T_1 already created and read an older version (v_1) of another page (P_1), T_1 needs to be rolled-back and restarted because of version inconsistency.

4.2 Discussion

In summary, DMV is a novel distributed multiversioning concurrency control algorithm that keeps only one physical copy of the application data at each node. It has the following features:

1. Update transactions are guaranteed to execute in parallel with any conflicting read-only transactions in two cases: i) master-only update transaction execution for multiple threads per node and ii) update-anywhere fully decentralized transaction execution with a single-thread per node.
2. DMV avoids aborts of read-only transactions in the common case by ordering the read-only transaction before a concurrent update transaction in the serialization order.

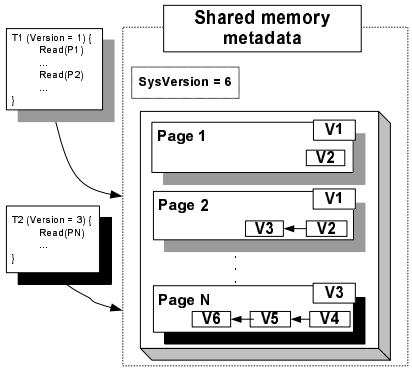


Figure 3. Concurrent "snapshot" creation of disjoint read-set transactions.

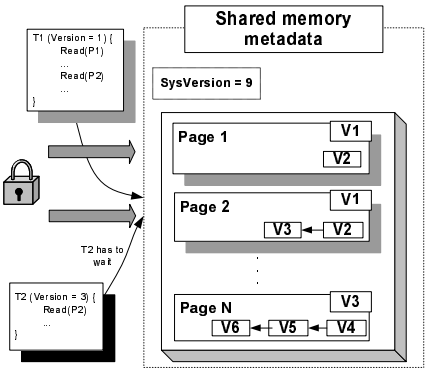


Figure 4. Overlapping read-sets with lock wait.

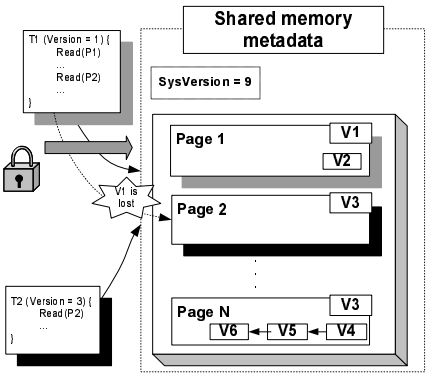


Figure 5. Overlapping read-sets with transaction rollback.

3. Read-only transactions with disjoint read sets always execute in parallel. Read-only transactions with overlapping read-sets can execute in parallel at the same replica if they require the same transactional system version or at different replicas if they require different versions.
4. For configurations that allow the use of a scheduler, version waits and aborts of read-only transactions due to version inconsistency are minimized through a version-aware scheduling technique.

One of the limitations of our systems is that it does not currently support the notion of predefined transaction order e.g., program order or application phases, such as for classic *barriers*. Since our system is version aware, these mechanisms could be easily added on top of DMV by enforcing an application-defined version order through version tags provided at `begin_transaction`.

Finally, since updates are fully replicated and minimal state is maintained outside of the transactional tier even in scheduler-based configurations, our transactional system provides the pre-requisites for easy failure reconfiguration. However, failure handling is beyond the scope of this paper.

5. Experimental Environment

In this section we introduce our benchmarks and we present the state-of-the-art asynchronous replication technique, Conflict-Aware replication, that we compare our solution against.

5.1 Distributed Hash Table Microbenchmark

We implement a classic hash table with resolving collisions through chaining. Each node declares the hash table in transactional shared memory space and issues local transactions with a specified probability of the transaction being a read-only (lookup) or read-write (insert, delete) transaction on random hash table items. We use a 128K hash table size and we vary the fraction of writes in the workload mix between 1% and 20%. The ratio between the complexities of read-only and update transactions is roughly 10 to 1.

5.2 TPC-W Benchmark

The TPC-W benchmark from the Transaction Processing Council (TPC) [30] is a transactional web benchmark for e-commerce systems. An e-commerce system consists of a front-end web server, an application server and a back-end database (see Figure 6). The (dynamic) content of the site is stored in the database. The client sends an HTTP request to the web server containing the URL of the script and some parameters. The web/application server executes the script, which issues SQL queries, one at a time, to the database and formats the results as an HTML page. This page is then returned to the client as an HTTP response.

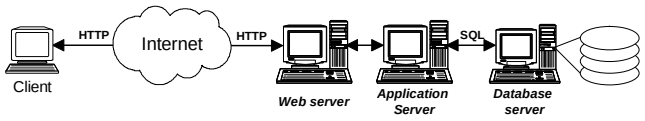


Figure 6. Common Architecture for Dynamic Content Sites.

The TPC-W benchmark simulates a bookstore. The database contains eight tables: customer, address, orders, order_line, credit_info, item, author, and country. The most frequently used are order_line, orders and credit_info, which contain information about the orders placed, and item and author, which contain information about the books. The database size is determined by the number of items in the inventory and the size of the customer population. We use the standard size with 288000 customers and 100000 books. The inventory images, totaling 180 MB reside on the web server.

We implemented the fourteen different interactions specified in the TPC-W benchmark document. Six of the interactions are read-only, while eight cause the database to be updated. The read-only interactions include access to the home page, listing of new products and best-sellers, requests for product detail, and two interactions involving searches. Update transactions include user registration, updates of the shopping cart, two order-placement transactions, and two for administrative tasks. The frequency of execution of each interaction is specified by the TPC-W benchmark. The most complex read-only interactions are BestSellers, NewProducts and Search by Subject which contain multiple-table joins.

The benchmark has three workload mixes characterized by different ratios of reads to writes. The browsing mix has 95% read-

only queries and 5% updates. The shopping workload, which is the one that most closely resembles a real-world scenario, consists of 80% read-only queries and 20% updates. The most update intensive workload is the ordering mix, which has 50% updates and 50% read-only queries.

5.3 Conflict-Aware Replication

In this section, we introduce a state-of-the-art asynchronous replication technique, *Conflict-Aware* replication [6, 7], for comparison with DMV. This technique has previously been proposed in the context of scaling database workloads, and in particular the workload of the TPC-W database back-end, on a cluster.

Conflict-Aware relies on a scheduler tier that distributes transactions on a database cluster to direct transactions in such a way to avoid conflicts. Conflicts are conservatively perceived at large granularities, such as, database tables. Each transaction explicitly pre-declares the tables it is going to access and their access type. The scheduler assigns the global serialization order of all transactions through per-transaction sequence [7] or version [6] numbers based on perceived conflicts between transaction table set pre-declarations. The total order thus assigned is enforced at all databases for all queries. The scheduler tags queries with the appropriate version number(s) for the tables they need to read. Each database replica keeps track of the completion of update transactions for each table. Queries whose per-table versions have not been produced yet are withheld in queues interposed in front of the database engine. The scheduler itself keeps track of versions of tables as they become available at each database replica and sends read-only queries to the database that has already produced the required versions. *Conflict-Aware* is thus a per-table distributed versioning technique where both write-write and read-write conflict resolution is per-table. In contrast, in DMV, conflict resolution is fine-grained (per-page). On the other hand, the perceived conflicts in Conflict-Aware are accurate, allowing the scheduler to avoid conflicts when scheduling reads. DMV in the master-update configuration may need to schedule read-only queries on replicas with conflicting versions due to lack of alternatives and may even abort read-only transactions in rare cases.

5.4 Experimental Setup

We run our experiments on a cluster of dual AMD Athlons with 512MB of RAM and 1.5GHz CPU, running the RedHat Fedora Linux operating system. The consistency unit we use is the memory page in all experiments. Our experiments focus on demonstrating the system scalability for the two benchmarks in the following respective configurations. We run the hash table microbenchmark in the *update-anywhere* cluster configuration. The application code is simple, hence easy to fully annotate for the purpose of operator overloading. Thus, access trapping in DMV is performed through operator overloading for both read and write accesses.

We run the e-commerce benchmark using the modified MySQL heap-tables issuing calls to the DMV transactional API in the *master-update* configuration. Access trapping in DMV is performed through a combination of memory protection violations and operator overloading. We compare DMV's scaling against the scaling of the state-of-the-art Conflict-Aware algorithm presented in section 5.3. Conflict-Aware runs on a platform with MySQL heap-tables enhanced with transactional semantics only, without DMV.

To demonstrate scaling for TPC-W, we run nine MySQL in-memory replicas on separate machines. We use 10 machines to operate the Apache 1.3.31 web-server, which runs a PHP implementation of the business logic of the TPC-W benchmark and use a client emulator, which emulates client interactions as specified in the TPC-W document. We use two configurations of MySQL in our in-memory tier (with and without indexes) to study the trade-off between individual node speed versus scalability over several replicas and the probability of aborts with different access patterns. To determine the peak throughput for each cluster configuration we run a step-function workload, whereby we gradually increase the number of clients from 0 to 1000. We then report the peak throughput in web interactions per second (WIPS), the standard TPC-W metric, for each configuration. At the beginning of each experiment, the master and the slave nodes memory-map an on-disk TPC-W database. We run each experiment for a sufficient time such that the data becomes memory resident and we exclude the cache warm-up time from the measurements.

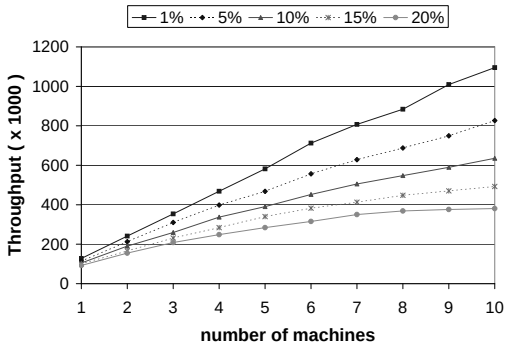


Figure 7. Throughput scalability with increasing fraction of update transactions.

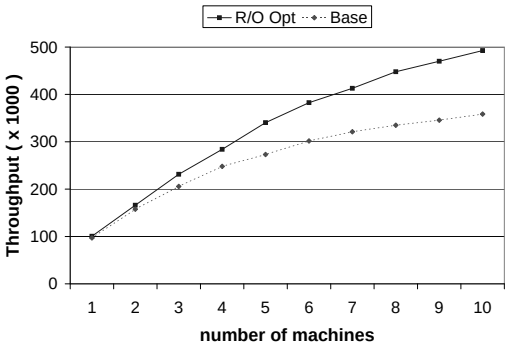


Figure 8. Increased scaling with read-only optimization versus without - for the case of 15% update transactions.

# of machines	1	2	4	6	8	10
% of aborts	0.00	0.57	1.69	2.94	4.05	5.08

Table 1. Percentage of aborts for the hash table microbenchmark with various cluster sizes.

6. Experimental Results

6.1 Distributed Hash Table Results

The graph in Figure 7 presents the throughput obtained for different percentages of update transactions (1%, 5%, 10%, 15%, 20%). The throughput represents the number of operations per second that the whole system achieves.

In Figure 8, we show the improvement due to optimizing aborts for read-only transactions for an intermediate case with 15% write transactions. The top curve shows scaling when we avoid aborts to read-only transactions by ordering them before a concurrent remote

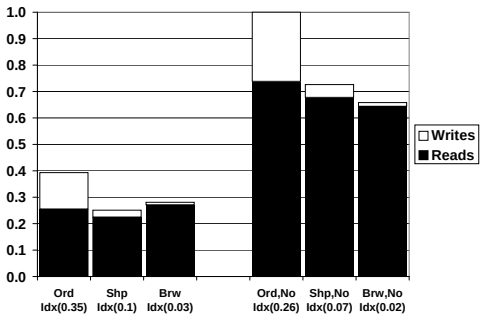


Figure 9. Relative query weights for MySQL heap tables in the two configurations (with and without indexes).

update transaction in the serialization order. The execution reflected in the bottom curve does not include this optimization and aborts a local read-only transaction whenever a conflicting diff flush is received at the node. We can clearly see that this optimization improves scaling. In Table 1 we plot the measured percentage of transaction aborts (transaction restarts) out of the total number of transactions executed for the same configuration with 15% write transactions, using the read-only optimization in DMV. As we can see from the graph, the fraction of aborts increases with the cluster size, but is overall low (5%). This is a result of the increase in the number of diff flushes received by each node with the increase in cluster size in our replicated system.

6.2 TPC-W Benchmark Results

6.2.1 Preliminary Experiments

The goal of this experiment is to compare the complexity of write versus read-only queries in each of the ordering, shopping and browsing TPC-W mixes. Since the read to write ratio affects scaling in replicated databases, we report results for two different database index configurations of our in-memory database system. The first configuration uses the same index set as in the on-disk InnoDB database we used in our prior work [7]. The second configuration uses no indexes, except for the primary keys. This configuration still conforms to TPC-W, since no particular index configurations are required or recommended by the specification.

For the purposes of this experiment, we run a workload session with only one emulated client, which submits 50000 TPC-W requests to a single database instance. Figure 9 depicts our findings, normalized to the total cost of the *ordering* mix in both the configuration with indexes (left) and without indexes (right). The white and black parts of each bar corresponds to the average cost in terms of average execution time at the database of write and read-only queries, respectively. The *Ord*, *Shp* and *Brw* abbreviations in the bar captions specify which workload mix the respective bar corresponds to. The numbers in the brackets denote the ratio of write versus read query cost for the particular configuration.

These experiments indicate that, with our in-memory database system, the cost of write transactions is significant. This problem is clearly shown in the ordering mix, where the cost of updates is the highest across all three mixes. The high cost of updates is caused by the cost of index update operations in our base system. Since our system builds upon the original MySQL heap table engine, we reused its *red-black tree (RB-Tree)* [12] index structure. The RB-Tree is a balanced binary search tree, which supports lookup operations with a constant $O(logN)$ cost. Insert and delete operations on the RB-Tree typically cause height imbalances, which then trigger tree rotations. The RB-Tree performs 2 rotations per write query on average and the cost of these rotations is significant in the in-memory database system.

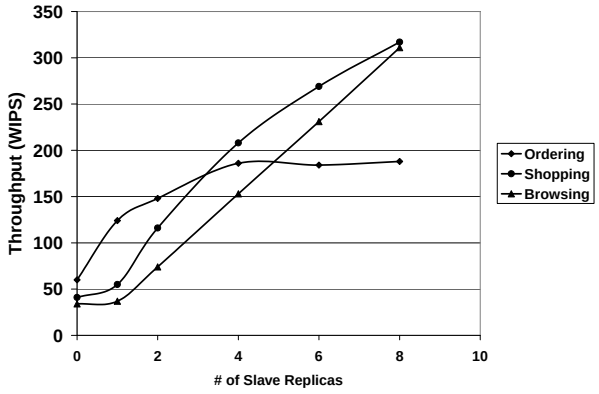


Figure 10. Throughput scaling in the database configuration with indexes for the browsing, shopping and ordering TPC-W mixes.

# of Slaves	Ordering	Shopping	Browsing
1	1.15%	1.44%	0.63%
2	0.35%	2.27%	1.34%
4	0.07%	1.70%	2.37%
6	0.02%	0.41%	2.07%
8	0.00%	0.22%	1.59%

Table 2. Level of aborts due to version inconsistency (indexed configuration).

In the configuration without indexes, (see Figure 9) the cost of read queries is relatively higher, hence we expect better scaling for our system in this configuration.

6.2.2 Scalability for the Configuration with Indexes

Figure 10 shows the throughput scaling obtained as we increase the number of slave replicas. We perform measurements with 1, 2, 4, 6 and 8 slave replicas, respectively.

The system exhibits close to linear scaling for the browsing and shopping mixes. The poor scaling of the ordering mix is caused by the fast saturation of the master replica with updates. The number of writes that the master executes increases with larger database clusters to sustain their higher overall throughput. Hence, workloads with a large fraction of writes, such as the ordering mix, saturate much sooner than read-mostly workloads due to the high average cost of writes in our system.

Table 2 shows the average number of read-only queries that needed to be restarted during the experiment due to version inconsistency. The numbers are presented as a percentage of the total number of queries that executed during the experiment. We see that the level of aborts is very low overall. Having more replicas generally helps version-aware scheduling by potentially increasing the number of version choices available. On the other hand, the fraction of updates executing at each slave, hence the frequency of version changes increases with the replicated cluster size. The abort rate is influenced by each of these factors in opposite directions but the positive effect of version-aware scheduling dictates a generally decreasing trend from 4 to 8 replicas.

To study this effect in more detail, Figure 11 shows the effect of our version-aware scheduling technique on abort rates for the 2, 4, 6 and 8 slave configurations, respectively, for the TPC-W browsing mix. The left bar in each configuration group represents the abort rate with plain load balancing and the right bar depicts the abort rate with version-aware scheduling. We can see that the effectiveness of

# of Slaves	Ordering	Shopping	Browsing
1	4.75%	2.05%	0.87%
2	3.83%	2.23%	1.50%
4	3.46%	3.90%	2.28%
6	2.88%	3.59%	1.74%
8	2.47%	3.48%	1.53%

Table 3. Level of aborts due to version inconsistency (non-indexed configuration).

version-aware scheduling generally increases with the number of data version choices at configurations larger than 4 replicas.

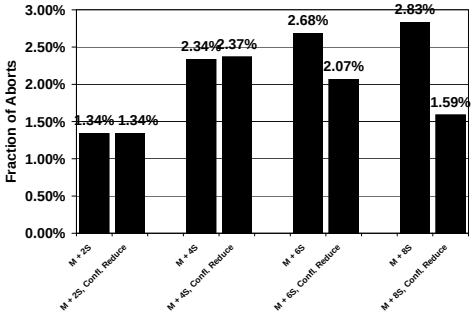


Figure 11. Abort rates for load balancing versus version-aware scheduling with increasing number of replicas.

6.2.3 Scalability for the Configuration without Indexes

In order to validate the scalability of the dynamic versioning algorithm with a different read to write cost distribution and different application access patterns, we ran the same experiments using the database without indexes. Figure 12 shows the throughput in this configuration.

These results show almost linear scalability for all the workload mixes. However, the throughput at the largest configuration is still lower than in the index configuration for all mixes. Table 3 lists the abort rates for the TPC-W database configuration without indexes.

For the ordering mix, the abort rate decreases steadily with an increase in the number of replicas while for the shopping and browsing mixes, the abort rate increases up to the 4-slave cluster configuration after which it gradually declines. The abort rates are slightly higher than in the indexed configuration for the ordering and shopping mixes, and comparable for the browsing mix. By performing full table scans, all read-only transactions, and in particular, complex queries such as BestSellers and NewProducts have similar read sets and access the maximum number of pages, hence the chance of version conflicts between read-only transactions is high. On the other hand, update transactions which are causing new versions to be produced are now much shorter and occur less frequently per unit of time in comparison, hence the non-uniform trend across mixes. Overall, the abort rates are acceptable in both configurations.

6.2.4 Comparison Against Conflict-Aware Asynchronous Replication

In the following experiment, we compare our *Distributed Multi-versioning* system against the state-of-the-art Conflict-Aware asynchronous replication scheme described in Section 5.3.

In Figure 13 we compare throughput scaling with the number of slave databases for our technique (DMV), and Conflict-Aware per-table versioning scheme depicted as *Conflict-Aware*, for the browsing, shopping and ordering TPC-W workload mixes. It can be seen

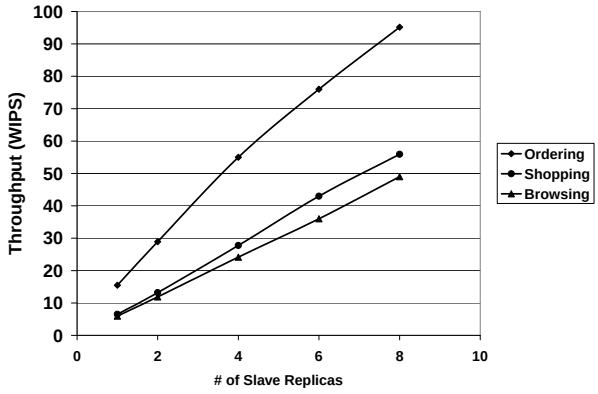


Figure 12. Throughput scaling in the database configuration with no indexes for the browsing, shopping and ordering TPC-W mixes.

that the DMV scheme has better performance scaling due to its finer-grained concurrency control scheme. In contrast, in Conflict-Aware, conflict resolution is per-table. Although read-only transactions execute in parallel with update transactions on a different replica, update transactions execute as a sequence of SQL queries at all replicas in the order pre-defined by the scheduler for any particular table. Hence, Conflict-Aware trades-off write-write concurrency and increased write execution duration (by executing SQL queries for writes sequentially per-table at all replicas) for improving read-write concurrency. This trade-off depends on the relative query weights of reads versus writes for the database system. The high cost of executing updates on our particular system disadvantages the Conflict-Aware approach.

However, DMV's main advantage is still its per-page conflict resolution for both types of conflicts, hence its good scaling. Furthermore, the advantage of DMV increases with a higher fraction of writes and the resulting increase in conflicts in the mix from browsing to ordering.

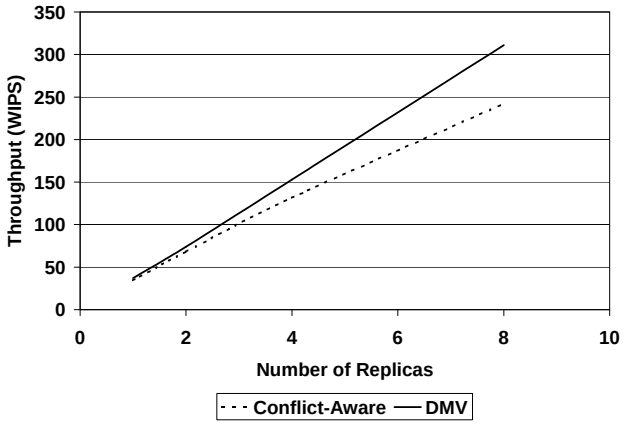
7. Related Work

DMV draws on related work in a variety of domains such as: persistent object stores, software distributed shared memory, transactional memory as a parallel programming paradigm and replicated databases. In the following, we can compare our system with only the few most relevant related works in each of these prolific systems areas.

DMV shares some of the goals of distributed persistent stores [22, 31, 10]. Some of these systems [31] focus on client-server paradigms, not on in-memory systems supporting transparent shared access. Argus [22] is one of the oldest systems to manipulate user-defined data dispersed across multiple, autonomous object stores called guardians. Their focus is, however, mostly on providing functionality for distributed collaborative data access and persistence rather than scaling applications on a cluster.

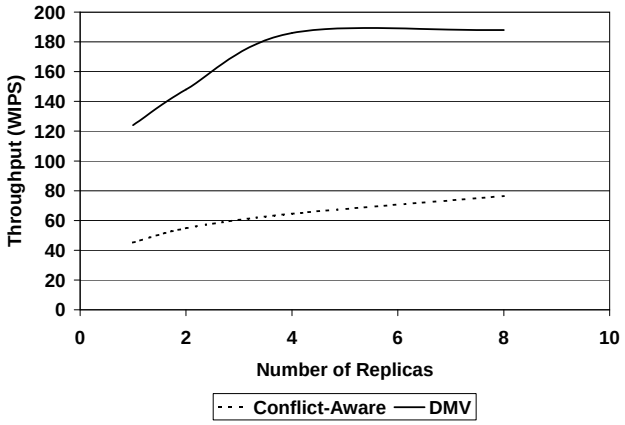
Our work is also related to recent research efforts on transactional memory [18, 17, 14, 16] and thread-level-speculation [27] in multiprocessor systems. To our knowledge this is the first technique that exploits the concurrency offered by distributed data versions in a transactional memory system.

Most SDSM libraries [20, 26, 28, 34], just like TreadMarks, have been used almost exclusively to support scientific parallel computation on networks of workstations. One of the few notable exceptions is recent work [29] leveraging the InterWeave SDSM system to provide a persistent store for distributed applications. Clients access the persistent store through a transactional RPC interface. The main difference is that their system supports several



(a) Browsing

(b) Shopping



(c) Ordering

Figure 13. Throughput scaling comparison against the conflict-aware scheduling technique.

weak consistency models instead of 1-copy serializability. A related trend is to support levels of consistency that are generally considered acceptable for transactional applications, such as *snapshot-isolation* [32, 21] and *session consistency* [13], or a continuum of consistency models with tunable parameters [33, 25]. On the other hand, for some applications, adjusting the application to a weaker consistency model may require non-trivial programmer effort or may cause user confusion.

A number of solutions exist for replication of relational databases that aim to provide both scaling and strong consistency. They range from industry-established ones, such as the Oracle RAC [2], to research and open-source prototypes, such as Distributed Versioning [6], C-JDBC [11], Postgres-R [19] and Ganymed [24]. The industry solutions provide both high availability and good scalability, but they are costly and require specialized hardware such as Shared Network Disk [2]. Some of the research prototypes use commodity software and hardware, but they either use coarse-grained concurrency control implemented in the scheduler [7, 6, 11] or rely on support for snapshot isolation inside the database [24, 32]. Recent replicated database systems explore providing snapshot isolation semantics on a database cluster [32, 21, 24]. None of these systems investigates general purpose transactional memory layers. Furthermore, these systems rely on the presence of unbounded numbers of *physical* copies at each distributed database node in order to provide scaling. Our system needs only one copy of data at each node, hence simplifies physical copy management. A direct performance comparison would be, however, instructive and is the objective of future work.

8. Conclusions and Future Work

In this paper we introduce a novel distributed concurrency control algorithm, *Distributed Multiversioning*, which preserves strong consistency and at the same time offers scaling. DMV exploits the naturally arising versions across transactional memory replicas in order to increase concurrency. In its most general form, DMV is an enhancement of a distributed shared memory protocol with transactional support in order to automatically detect and resolve conflicts caused by data races. DMV improves scaling by running conflicting read-only and update transactions on different nodes in order to avoid conflict waits and transaction roll-backs in the common case. Wherever possible, a version-aware scheduling algorithm strives to distribute read-only transactions requesting different version numbers for their items across different replicas. Our techniques allows us to avoid overheads associated with maintaining multiple copies in systems that use multiversioning techniques inside a single node. We have shown that our distributed concurrency control algorithm provides scaling for both i) a generic C++ application accessing shared memory data structures in a fully distributed manner and ii) a recognized transactional application which can benefit from version-aware scheduling.

In our future work, we will extend our scheme to hybrids between limited local multiversioning and distributed multiversioning and compare our system with traditional multiversion concurrency.

Acknowledgments

We thank the anonymous reviewers and the systems and compiler groups at University of Toronto for their detailed comments and suggestions for improvement on the earlier versions of this paper. We further acknowledge the generous support of the Natural Sciences and Engineering Research Council of Canada (NSERC), IBM Centers of Advanced Study (IBM CAS), Ontario Centers of Excellence (OCE) and Canadian Foundation for Innovation (CFI).

References

[1] Mysql Database Server. `http://www.mysql.com/`.

[2] Oracle Real Application Clusters 10g. `http://www.oracle.com/technology/products/database/clustering/`.

[3] Transactional web e-commerce benchmark. `http://www.tpc.org/tpcw/`.

[4] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[5] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems*, March 2003.

[6] C. Amza, A. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *ACM/IFIP/Usenix International Middleware Conference*, June 2003.

[7] C. Amza, A. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *Proceedings of The 21st International Conference on Data Engineering (ICDE 2005)*, April 2005.

[8] The Apache Software Foundation. http://www.apache.org/.

[9] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[10] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA*, 2003.

[11] E. Cecchet, J. Marguerite, and W. Zwaenepoel. RAIDb: Redundant array of inexpensive databases. In *IEEE/ACM International Symposium on Parallel and Distributed Applications (ISPA'04)*, December 2004.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[13] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *20th International Conference on Data Engineering, Boston, Massachusetts*, April 2004.

[14] K. Fraser. Practical lock-freedom. In *Ph.D. Thesis, King's College, University of Cambridge*, 2003.

[15] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[16] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of OOPSLA*, 2003.

[17] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *International Conference on Principles of Distributed Computing (PODC)*, 2003.

[19] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *The VLDB Journal*, pages 134–143, 2000.

[20] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[21] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of SIGMOD*, 2005.

[22] B. H. Liskov and R. W. Scheifler. Guardians and actions: linguistic support for robust, distributed programs. In *ACM Transactions on Programming Languages and Systems, 5:381-404*, 1983.

[23] PHP Hypertext Preprocessor. http://www.php.net.

[24] C. Plattner and G. Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proceedings of the 5th ACM/IFIP/Usenix International Middleware Conference*, October 2004.

[25] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. FAS - a freshness-sensitive coordination middleware for a cluster of OLAP components. In *VLBD*, 2002.

[26] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A low overhead software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[27] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[28] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software coherent shared memory on a clustered remote write network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 170–183, October 1997.

[29] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott. Integrating remote invocation and distributed shared state. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[30] Transaction Processing Council. http://www.tpc.org/.

[31] S. J. White and D. J. DeWitt. Quickstore: A high performance mapped object store. In *SIGMOD*, 1994.

[32] S. Wu and B. Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *Proceedings of the 21st International Conference on Data Engineering*, April 2005.

[33] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 305–318, October 2000.

[34] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation*, pages 75–88, November 1996.