Extending Distributed Shared Memory with Transactional Memory Support

Sven Lundgren
Abstract

Extending Distributed Shared Memory with Transactional Memory Support

Sven Lundgren

Parallel programming has become increasingly important both as a programming skill and as a research topic over the last decades. Multicore computers and mobile devices are part of our daily lives, and computer clusters facilitate research in many areas all over the world. Distributed shared memory (DSM) is the prevalent programming paradigm in cluster computing, allowing programmers to address all available memory on a computer cluster as one large, contiguous address space. Even with the support of DSM, data races pose a major hurdle for programmers in most applications, and existing lock-based concurrency mechanisms are complicated to use and not suitable for all tasks.

Transactional memory (TM) places the burden of coherence and concurrency on the system rather than on the programmer, leaving the programmer with the simple task of defining a transactional code block. As an attractive alternative to lock-based synchronization, it also shifts the burden of code optimization towards the system rather than the programmer. Transactional memory has traditionally relied on centralized concurrency protocols inherently unsuitable for large distributed settings, and therefore the question whether transactional memory can be implemented in a scalable manner suitable for large distributed systems can be asked.

In this thesis, a transactional memory extension of distributed shared memory is presented and compared as an alternative to lock-based synchronization. A synthetic random access algorithm shows significant throughput scaling for up to at least 256 cores across 32 DSM nodes, with transactional memory outperforming cohort locking for all cases above 16 cores. The benefits of local decision making and a distributed coherence protocol are also shown to be of utmost importance.
Acknowledgements

I would like to thank my supervisor Stefanos Kaxiras and my reviewer Konstantinos Sagonas, both at Uppsala University (Department of Computer Science), for the opportunity to pursue this thesis at Uppsala University and for their help and support throughout the process.

I would also like to thank Magnus Norgren and David Klaftenegger for their invaluable help with technical details and answers to countless questions.
# Contents

1 Introduction 11

2 Background 13

- 2.1 Distributed Shared Memory . . . . . . . . . . . . . . . . 13
- 2.2 Distributed Communication . . . . . . . . . . . . . . . . 14
- 2.3 ArgoDSM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
- 2.4 Transactional Memory . . . . . . . . . . . . . . . . . . . . . . 15
- 2.5 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

3 Design and implementation 19

- 3.1 Transactional interface . . . . . . . . . . . . . . . . . . . . . . 20
- 3.2 Ownership records . . . . . . . . . . . . . . . . . . . . . . . . 21
- 3.3 Transaction metadata . . . . . . . . . . . . . . . . . . . . . . . 22
- 3.4 Transaction commit . . . . . . . . . . . . . . . . . . . . . . . . 23
  - 3.4.1 Read validation . . . . . . . . . . . . . . . . . . . . . . . . 23
  - 3.4.2 Write commit . . . . . . . . . . . . . . . . . . . . . . . . . 24
  - 3.4.3 Aborting and restarting transactions . . . . . . . . . . . 25
- 3.5 Initial design overview . . . . . . . . . . . . . . . . . . . . . . 25
- 3.6 Initial design flaws . . . . . . . . . . . . . . . . . . . . . . . . 27
- 3.7 Selective self-invalidation . . . . . . . . . . . . . . . . . . . . 28
- 3.8 Ownership record cache . . . . . . . . . . . . . . . . . . . . . . 28
- 3.9 Distributing coherence . . . . . . . . . . . . . . . . . . . . . . . 30
  - 3.10 Final design overview . . . . . . . . . . . . . . . . . . . . . . 31
    - 3.10.1 Ownership record cache . . . . . . . . . . . . . . . . . . 31
    - 3.10.2 Read validation . . . . . . . . . . . . . . . . . . . . . . . 31
    - 3.10.3 Write commit . . . . . . . . . . . . . . . . . . . . . . . . 31
    - 3.10.4 Optimized example . . . . . . . . . . . . . . . . . . . . . 32

4 Evaluation 35

- 4.1 Evaluation environment . . . . . . . . . . . . . . . . . . . . . . 35
- 4.2 Evaluation algorithm . . . . . . . . . . . . . . . . . . . . . . . . 35
- 4.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
  - 4.3.1 Low write load . . . . . . . . . . . . . . . . . . . . . . . . . 38
  - 4.3.2 Medium write load . . . . . . . . . . . . . . . . . . . . . . . 39
  - 4.3.3 High write load . . . . . . . . . . . . . . . . . . . . . . . . . 40

5 Conclusion 43

- 5.1 Problems and limitations . . . . . . . . . . . . . . . . . . . . . 43
- 5.2 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

Bibliography 47
List of Figures

1 A two tier design problem faced when programming in a cluster environment. ........................................... 13
2 A simple illustration of the functionality of a DSM System. . . 14
3 Two transactions competing for the resource X means that only one can complete in order to maintain data consistency. . . . . 16
4 A single ownership record as implemented in the TM system. . . 21
5 A detailed view of the metadata struct. The Write Copy ptr is naturally only present in the writeset. ....................... 22
6 The commit phase as seen from both user and system perspective. 23
7 Prototype design of the Transactional Memory system. . . . . . 27
8 One ownership record cache entry as implemented in the TM system. ......................................................... 29
9 Optimized design of the Transactional Memory system. . . . . 31
10 Results of simulation under low write load on linear scale. . . . 38
11 Results of simulation under low write load on logarithmic scale. . 38
12 Results of simulation under medium write load on linear scale. . 39
13 Results of simulation under medium write load on logarithmic scale. .......................................................... 39
14 Results of simulation under high write load on linear scale. . . . 40
15 Results of simulation under high write load on logarithmic scale. 40
Introduction

Since the turn of the century, parallel computing has developed from something mostly used in high performance computing to a necessity even in our home computers and mobile devices. With power consumption and heat generation limiting frequency scaling, the primary solution to faster computers is no longer to increase processing unit frequency but instead to add more transistors and subsequently more processing units [1]. For some tasks, a single symmetric multiprocessor or chip multiprocessor may not provide enough processing power or memory necessary to hold all related data. Using a number of processing units each with their own associated memory connected to form a computer cluster, more processing power and memory can be obtained at the cost of increased programming complexity. A distributed shared memory (DSM) system, providing a coherent global address space across all processing units can help to reduce programming complexity in a distributed environment, but data coherence is still threatened by data races and must be dealt with by the programmer through provided synchronization mechanisms. Finally, in order to achieve scalability in a distributed system, it is critical to make decisions locally and avoid expensive remote communication [2], and poor use of provided concurrency mechanisms may cause severe scaling issues.

If one was to place the burden of coherence and concurrency on the under-lying distributed shared memory system instead, data coherence could be guaranteed at the same time as heavily reducing programming complexity. Transactional memory (TM) by definition puts the burden of coherence and concurrency on the runtime system, and leaves the programmer with the simpler task of defining a transaction. However, transactional memory in a distributed setting relies on remote communication in order to validate and commit transactions, and therefore the question whether a scalable transactional memory system can be implemented in a distributed setting can be posed.

In this thesis, a fully optimistic distributed software transactional memory system is presented and implemented on top of a software DSM system titled ArgoDSM [2]. In order to achieve scalability, a distributed implementation of transactional memory needs to promote concurrency among transactions and eliminate as much system overhead as possible. Transactions must also work to minimize expensive remote communication. To achieve this goal, two main principles are proposed. Transactional read accesses follow the principle of read validation (RV) by optimistically reading data without locking access to it, and instead validate that no changes have been made to read data at transaction
commit time, self-aborting only if any discrepancy is detected. Transactional writes work on a local copy of cached data, and only attempt to write back data remotely at commit time. A transaction must commit atomically, and therefore exclusive access to any data written must be ensured, specifically by obtaining exclusive write access to all write locations at commit time. As opposed to obtaining exclusive access eagerly at write access detection, this lazy acquire (LA) principle is employed to increase concurrency between transactions, and in order to validate transactional reads and ensure atomicity for transactional writes, remote data must be version controlled and locked through an ownership record (orec) mechanism.

In addition to these principles, a key idea of this work is to use locally cached, by the underlying DSM, values of remote data and ownership records in order to optimistically perform transactions. Finally, data and metadata prefetching and cooperation between transactions running on the same node ensures that cached values are reasonably up to date.

After implementing the transactional memory system adhering to the ideas and principles mentioned above, we show the importance of local decision making and a distributed coherence protocol without software handlers by comparing an initial prototype failing to implement these features with a final implementation.

The remainder of this thesis is structured as follows. In Chapter 2, background on relevant topics and related work is presented. A prototype design is proposed in Chapter 3 along with various optimizations culminating in a final design, with performance and scaling of both prototype and final design along with each optimization displayed in Chapter 4. Finally, in Chapter 5 various problems, limitations and remaining future work is discussed.
2 Background

2.1 Distributed Shared Memory

Modern computers and mobile devices alike are today almost exclusively equipped with multiple processing units. The chip multiprocessor (CMP) comes with many advantages for the user, but for the programmer it presents the problem of dealing with concurrent access to shared memory locations such as primary and secondary memory, but also to faster memory such as shared caches. In the same fashion, a cluster of interconnected CMPs creates another layer of complexity as each CMP by default has direct access to its own hardware memory, but can not trivially access the remote memory of other CMPs. While shared memory programming can be applied on a chip level, remote memory accesses are typically done through message passing as illustrated in Figure 1. Even with the support of a standardized message passing interface (MPI), programming using this two tier design can be a daunting task for many programmers.

[Diagram showing a two-tier design problem in a cluster environment]

**Figure 1:** A two tier design problem faced when programming in a cluster environment.

Distributed shared memory (DSM) is a form of memory architecture where physically separate memory can be addressed as one logically shared address space. The physical network layout and its resulting two tier design is abstracted away by the DSM System as illustrated in Figure 2, and programming is similar to that of shared memory on one CMP. A DSM system can be implemented either in hardware or in software, and, as this thesis targets a software
The main difficulty to overcome in a distributed system is the latency incurred by communication between remote nodes. In the past, network latency and bandwidth were orders of magnitude worse than corresponding memory latency and bandwidth, resulting in overhead by software message handler execution being relatively insignificant in a distributed network environment. Coherence protocols were designed to hide network latencies through relaxed consistency models [3] and to minimize communication bandwidth. Home-centric systems, where a single node handles concurrency and coherence through a request and response model, were simple to implement and prevented data races from threatening coherence. However, rapid advances in network and hardware technology over the last few decades, specifically network latency closing the gap on CPU latency, point to the need for a new design philosophy in order to scale shared memory in distributed systems [2]. Specifically, more bandwidth should be traded in order to reduce latency, message handler execution should be minimized as well as data movement for critical sections. This philosophy renders home-centric system design unsuitable in large distributed environments, and instead points to a truly distributed design philosophy, where no central point of coherence can create a bottleneck in the system.
2.3 ArgoDSM

ArgoDSM [2] is a software distributed shared memory implementation with emphasis on enhancing parallel scaling by performing as many decisions as possible locally in order to reduce synchronization overhead. ArgoDSM presents a novel coherence protocol suitable for data-race-free programs flipping the way coherence and critical sections are traditionally done motivated by advances in hardware and network technology. A key concept is to trade more bandwidth (through for example prefetching) for reduced latency (by eliminating message handlers). Self-invalidation and self-downgrade distribute coherence control so that each node makes decisions locally without communication, and a novel home-node directory protocol servicing remote direct memory access (RDMA) reads and writes initiated by the requesting core ensures a message-handler-free protocol. Finally, an extended queue delegation locking algorithm [4, 5] ensures minimal data movement for critical section execution. Operating in user-space, ArgoDSM offers scaling of pthreads programs from one to thousands of nodes with minimal effort.

2.4 Transactional Memory

Transactional memory is an alternative to lock-based synchronization in which a number of load and store operations attempt to execute as one single atomic operation. Transactional memory is inherently optimistic as multiple transactions can run in parallel until a conflict occurs. A conflict is defined as two or more transactions attempting to access the same resource simultaneously, often restricted to the case where at least one transaction attempting to alter the resource, as illustrated in Figure 3. Coherence between multiple transactions is guaranteed by definition as only one transaction may proceed if a conflict occurs, while all remaining transactions involved in the conflict must abort or restart. Finally, a transactional memory system must adhere to the principles of atomicity, consistency and isolation on a transactional level. Atomicity ensures that all stores of a transaction are committed or discarded as a unit. Isolation dictates that no stores can be visible outside of a transaction until the transaction commits. Finally, consistency ensures that memory operations in a transaction take place in order.

Locks and critical sections are pessimistic by definition, prohibiting concurrency where there is a risk of conflict. Furthermore, lock-based synchronization can be difficult to implement efficiently and the risks of deadlocks is left entirely up to the programmer to deal with. The main incentive for transactional memory is to simplify concurrent programming in comparison to lock-based synchronization by shifting the burden of coherence to the Transactional Memory System. Transactional memory is deadlock free, and its performance is based on the system implementation and the workload suitability rather than user implementation. In a traditional transactional memory implementation, a code block is defined as transactional by the programmer, and the underlying transactional memory system transparently guarantees atomicity, consistency and isolation for all operations in the code block.

Transactional memory can be implemented in both hardware and software, and while initial research focused on implementations in hardware, in 1997
Shavit and Touitou extended the concept to support implementations entirely in software [6]. With the focus of processor development shifting from fast single core chips to multicore chips in the early 2000’s, software transactional memory gained significant research interest as an alternative concurrency mechanism to lock-based synchronization. A number of STM implementations have received significant publicity such as Transactional Locking II [7], DSTM2 [8], McRTSTM [9] and RingSTM [10], but the vast majority target chip multiprocessors (CMPs) and do not function or scale well on modern distributed computer clusters. In contrary to cache coherent CMPs, where access to shared memory is typically fast and relatively local, large-scale computer clusters suffer from expensive communication to memory on remote nodes and therefore a different design philosophy is required. The following section explores and discusses existing Software DSM implementations.

2.5 Related Work

In an attempt to leverage the properties of CMP nodes in a cluster, researchers at the University of Manchester developed a STM system titled DiSTM [11] based on the DSTM2 transactional engine. DiSTM allows for one transaction per thread to execute concurrently on each node, and comes pre-packaged with one decentralized and two centralized coherency protocols. Transactions are executed on thread-private clones of objects, and upon commit a contention manager detects and resolves conflicts with concurrent transactions. However, the coherence protocols provided in DiSTM rely either on system-wide broadcast with a global ticketing mechanism, or on serialization leases acquired from a master node. Serialization leases, in contrast to broadcasting, attempt to minimize communication over the network but instead cause a bottleneck in the system, especially as more nodes are added and all communicate with the master node on acquire and release of each lease. Benchmark results indicate that the system in general does not scale over 32 threads [11], which indeed severely limits the systems usability on modern large-scale clusters.

Manassiev et al. [12] instead implement a distributed approach of multiversioning, employing full data-set replication where one version of each page is stored on a separate node. A master node coordinates transactions by delegating execution of read transactions to nodes containing desired page versions, allowing read transactions to concurrently execute without interference from write transactions. Write updates are serialized on the master node, which

\[ \text{Figure 3: Two transactions competing for the resource X means that only one can complete in order to maintain data consistency.} \]
subsequently broadcasts results to all slave nodes. Slave nodes only apply updates when required by an executing transaction, resulting in minimal delay on update broadcast. However, a centralized master node once again provides a bottleneck limiting scalability, and while read dominated workloads show close to linear scalability for up to eight slave nodes, write dominated loads show limited scalability and write performance would likely degrade further should more slave nodes be added.

In order to overcome both master node bottlenecks and network congestion, a multi-versioned distributed STM system based on the paradigms of database replication was proposed and implemented as D²STM [13]. Built upon JVSTM [14] providing versioned history of values, the system leverages an atomic broadcast algorithm to enforce update serialization and full data set replication on all nodes in the cluster. In D²STM, atomic broadcast of committing transactions ensures that ongoing transactions conflicting with a currently committing transaction need not abort, but merely apply the updates of the committing transaction before attempting to commit themselves. Versioned boxes are provided by the underlying multi-versioned JVSTM, allowing read transactions to proceed in spite of conflicting write transactions and rarely, if ever, have to abort. In spite of its distributed certification scheme (Bloom Filter Certification) exploiting space efficient Bloom Filter-based encoding, multi-versioned algorithms in general target read-heavy workloads. Additionally, full data set replication along with versioned boxes may limit the usability of the system due to space constraints. Finally, scalability was only shown for up to four threads each on eight nodes [13], while modern clusters are able to run thousands of threads in parallel.

While multi-versioning algorithms tend to favour read-heavy workloads, an interesting result is achieved by Kim et al. [15], where a prototype implemented in the DTM framework HyFlow [16] enhances concurrency among write transactions in multi-versioning DTM systems by prioritizing commutative write transactions. Two transactions are commutative if the state of an object is the same regardless of the order in which the transactions are applied. Furthermore, the commutative request first (CRF) scheduler alternates between two epochs, one in which only commutative transactions are allowed to proceed, and one in which the remaining non-commutative transactions run. Benchmarking for up to 120 threads on 10 nodes indicates that the system scales significantly better with up to five times higher throughput for both write- and read-heavy loads than its main competitor DecentSTM [17].

All of the above mentioned systems suffer from at least one major drawback as none of them is proven to scale to a large number of processors, whether the cause is network congestion, master node bottlenecks or space constraints. In an effort to target larger-scale clusters, Bocchino et al. developed Cluster-STM [18] scaling efficiently up to at least 512 processors mainly by aggregating communication overhead with communication of actual data. Bocchino et al. realized that transactional memory is inherently fit for the PGAS (Partitioned Global Address Space) programming model, and aimed towards creating a highly scalable transactional runtime system based on software transactions, as hardware support for shared address spaces across nodes is uncommon. The key ideas of Cluster-STM is to minimize the extra operations imposed by the STM, aggregate communication and exploit locality. Contrary to previously mentioned
Distributed STM systems. Cluster-STM does not run in a JVM nor does it rely on an underlying DSM implementation, but instead leaves issues such as caching of data up to the application programmer. Cluster-STM also compares four different approach mixtures of read locking and validation, early and late acquire for writes, and finally undo logging and write buffering. Interestingly, results show that the system scales competitively in comparison with locks for the provided benchmarks, and that both read locking and write buffering can be worth considering in a distributed environment.
With the experience from previous sections in mind, an initial design prototype can now be presented. The implementation of transactional memory is intended to extend ArgoDSM with transactional memory support, and it is therefore natural to base key design decisions on the ArgoDSM philosophy detailed in Section 2.3. As such, the key concepts of trading bandwidth for latency and avoiding software message handlers follow. The latter can be achieved by exploiting ArgoDSM’s one-sided RDMA capabilities. Additionally, avoiding a centralized coherence protocol is essential for scaling in large distributed environments as explained in Section 2.2.

Before presenting the initial prototype design, a few assumptions have to be made. First of all, in order to be fully compatible with ArgoDSM, it is important that user space data is either treated as transactional or non-transactional. This means that user space data that is considered transactional will no longer be guaranteed coherent under ArgoDSM’s standards, while user space data that is written through ArgoDSM will no longer be guaranteed coherent under transactional standards. Furthermore, it is assumed that each hardware cluster node involved in a transactional program executes at least one instance (software node) of ArgoDSM. Finally, transactional memory is inherently fit for applications that do not expect many transactional conflicts but still has to guard against possible conflicts. This follows from the fact that a transactional application will approach serial execution once conflict rate approaches 100%, and as detailed in Sections 3.4.1 and 3.4.2, an aborted and restarted transaction will trigger additional remote cluster communication while non-conflicting transactions, especially transactions consisting only of read operations, will trigger little remote communication. Therefore, it makes sense to design transactional memory by targeting and optimizing for the expected use-case, rather than for all possible cases. In other words, this means that an optimistic approach is desirable, in which conflicts are not expected and first dealt with when they happen rather than prevented from happening.

In Sections 3.1–3.4 each core concept of the design is explored followed by a prototype overview in Section 3.5. Section 3.6 discusses various flaws present in the prototype design, and Sections 3.7–3.10 explore numerous important optimizations in order to reduce remote communication and align the prototype design with the ArgoDSM philosophy.
3.1 Transactional interface

The transactional interface provides users with the ability to perform virtually any task in transactional form. A transaction is defined by the user as an unnamed function in the form of a C++ lambda expression. Each instruction within the lambda expression is treated as part of the transaction. As this implementation extends software DSM already utilizing memory access operator overload, transactional memory read (load) and write (store) operations are explicitly called by the user. A very simple transactional code block is shown in Listing 1.

```c
transaction::init(); /* Initialize the TM system */
/
/* Run transaction, usually done in parallel through pthreads */
atomic_i_tx([&]() { /* Begin transactional code block */
    val = read_tx(&array[x]); /* read the value at array index x */
    val++; /* increment the read value */
    write_tx(&array[x], val); /* write back val to array index x */
}); /* End the transactional code block */
/
transaction::finalize(); /* Clean up */
```

Listing 1: Basic transactional code block, for clarity surrounding code and declarations are omitted.

The transactional memory system is initialized with a call to `transaction::init()` performing necessary initialization, and finalized with a call to `transaction::finalize()` cleaning up all memory allocated by the transactional memory system. To initiate a transactional code block, the lambda expression `atomic_i_tx([]() {...})` is called, with any transactional code contained between the inner brackets representing an anonymous function. The lambda expression takes a capture list `[]` and an argument list `()`. A transactional read operation is made through `read_tx(&addr)` and returns the value referenced by `addr`, while a transactional write operation is made through `write_tx(&addr, val)` using `val` as intended write value on target address `addr` without return value. Transactions are restarted automatically if necessary upon conflict with another transaction, but if the user wishes to manually restart or abort a transaction in user code, `restart tx()` and `abort tx()` provide the respective functionality.

Coherence is offered on byte level as a natural extension of ArgoDSM. This means that any data type is handled, although transaction concurrency is offered on word level in an effort to minimize the memory overhead introduced by transactional memory. As a result of this design choice, transactions operating on data types smaller than one word or concurrently on very large data structures, while coherent, may experience reduced efficiency due to transactional conflicts. It is good practice to divide a large problem into many transactions with few instructions in each, instead of executing many instructions in one large transaction.

As this thesis targets distributed systems, running transactional code on one thread or node as depicted in the very basic example above serves little purpose. Pthreads is the preferred way to distribute data-race-free workload to multiple threads on a software node in ArgoDSM, and transactional memory presents an easy way to ensure that the provided workload is indeed data-race-
free. For more information about how to set up and correctly parallelize a program for ArgoDSM, refer to the quickstart and tutorial on the ArgoDSM GitHub repository page. Finally, it is worth mentioning that I/O and other side effects are not handled by the transactional memory system, and should in most cases be executed outside of transactions. If executed within a transaction, such actions will be performed in place on every restart of a transactional code block leading to possibly unwanted consequences, which is further discussed in Section 5.2.

3.2 Ownership records

By defining that a transaction upon conflict with another transaction should be able to decide whether to continue or restart on its own, without the use of software message handlers or a centralized coherence mechanism, one can ensure that non-conflicting transactions do not impede the performance of other transactions. In order to achieve this, it is essential that good decisions can be made based on local knowledge. We borrow the popular concept of ownership records (orecs) from the original paper on STM [6] and make a number of alterations to suit a distributed implementation. An ownership record is defined to consist of a simple lock and a version counter, and is implemented as a single 64-bit unsigned integer with the LSB as lock-bit and the remaining 63 bits as version counter, illustrated in Figure 4. To prevent version counter overflow, once the maximum number the version counter can hold has been reached it is reset to 0 again. A 64-bit unsigned integer was chosen over a 32-bit unsigned integer in order to reduce the likelihood that a version counter wraps around and falsely returns to its first encountered state, which is considerably less likely with a maximum version counter of $2^{63}$ rather than $2^{31}$.

![Figure 4: A single ownership record as implemented in the TM system.](https://etascale.github.io/argodsm/)

By allocating an array of ownership records in DSM memory, ownership records can be read and written through RDMA operations facilitated by the DSM system. Ownership records control both transaction concurrency and transactional data coherence. Each ownership record governs access to one or more DSM data locations through a simple address hash translation on the DSM memory address. An ownership record must be acquired (locked) by a transaction intending to write to a data location governed by it. This is done through an atomic compare and exchange operation, guaranteeing that only one transaction can acquire an ownership record at a time. Any transaction can therefore inspect an ownership record at any time through a simple DSM read and determine if another transaction is currently writing to data guarded by it. Once a transaction successfully writes to a DSM data location, it must both release

---

1https://etascale.github.io/argodsm/
access to its held ownership records and increment each held ownership record’s version counter. In order to detect a transactional conflict, a transaction can compare the version counter of an ownership record between the start and the end of the transaction. If the version counter differs between the two points in time, another transaction has written to DSM data guarded by the ownership record, and the current transaction must abort and restart. ArgoDSM guarantees atomicity between DSM read and atomic write operations, which is why only the write operation needs be atomic.

### 3.3 Transaction metadata

As only one transaction can be executed in one thread at a given time, a straightforward way of ensuring that each transaction has its own set of metadata is to use *Thread Local Storage*. Each thread by definition has its own copy of the transaction metadata structs that exist for the full life span of the thread. The *readset* and *writeset* metadata structs both extend a base class named `setbase`, providing a templated `std::set` storing information on read- and write accesses made by the transaction along with collecting various transaction statistics. A detailed view of how the writeset map entry is designed can be seen in Figure 5.

![Figure 5](image_url)

*Figure 5: A detailed view of the metadata struct. The Write Copy ptr is naturally only present in the writeset.*

The metadata structs are populated by the transactional interface read and write instructions. Each time a read or write instruction is called through user code, the calling thread will collect and store metadata in its respective set, with the DSM data address as key and related metadata as value. In addition, the writeset stores a pointer to a copy of the DSM data accessed by a transactional write. All transactional data written by the transaction is stored in this copy until transaction commit time, which ensures fulfillment of the isolation property.

If a set already contains metadata on a requested address and size, no duplicate entry will be made. Additionally, in order to optimize performance, the writeset copies metadata from the readset if an entry on the requested address and size already exists in the readset. Similarly, the readset refrains from creating a new entry for an address if one already exists in the writeset, and instead returns a pointer to the data copy held by the writeset in order to reflect changes made by the transaction. Why this is possible is further explained in Section 3.4. Finally, each set implements the respective read validation and write commit.
3.4 Transaction commit

Arguably the most important and most difficult part of a transactional memory system is how to remotely validate and commit a transaction. At the same time as ensuring correctness, a large part of the transactional memory system performance relies on efficient validation and commit procedures.

Once all user code in a transactional code block has executed, the transactional memory system will attempt to finalize, also referred to as commit, the transaction. Transaction commit consists of two phases: read validation and write commit. Only if a transaction passes read validation may it attempt to commit writes, and a transaction can only be regarded as concluded after successful write commit. An overview of the transaction commit phase can be observed in Figure 6.

![Figure 6: The commit phase as seen from both user and system perspective.](image)

3.4.1 Read validation

Read validation is the concept of validating that any transactional read operations have not been subject to a conflict in which they should abort during the course of the transaction. This is done by comparing each ownership record’s lock bit and version counter first recorded during transactional reads with their remote counterparts. Validation can only be considered successful if no lock is held or version counter discrepancy is found. Correctness of a successful validation is ensured by the fact that a transaction can be considered as one atomic operation, and therefore it is sufficient to ensure that no external changes to data involved in the transaction have been made between its first encounter
during the transaction and any point after all user code has been executed. In other words, it is sufficient to check for alterations between stored ownership record values and their remote counterpart at any point during transaction finalization, as seen in Figure 6.

Read validation iterates through each readset entry. It aims to exploit the DSM coherence protocol in order to minimize the amount of remote operations performed by using locally cached values of the remote ownership record. However, since validation requires up to date data, at least to the point where transaction finalization begins, this means that validation also has to be preceded by invalidation of any locally cached DSM page corresponding to the entry in the readset in order to ensure that it is up to date. After invalidating the cached DSM data, the ownership record values for the readset entry are compared to the remote values by a simple DSM read triggering a remote fetch of the page containing this ownership record. If all entries are validated successfully, the validation phase is considered successful. If validation of any ownership record fails, the whole validation phase fails. The prototype implementation of the validation phase is shown in Algorithm 1.

success = true;
Invalidate DSM data;
foreach readset entry do
  if success == false then
    Invalidate DSM data;
    break;
  end
  foreach index orec pair do
    if local orec != remote orec then
      success = false;
      break;
    end
  end
end
return success;

Algorithm 1: The read validation phase of transaction commit.

3.4.2 Write commit

Write commit is the concept of exposing changes made locally by a transaction to the entire system. Contrary to validation, a DSM write containing the local changes is not sufficient to preserve the atomicity property of transactional memory. This is the most critical part of this transactional memory system, as correctness has to be ensured without heavily impeding performance. It is critical from a performance standpoint as read-only transactions, that do not employ write commit at all, can never cause another transaction to abort, whereas write transactions require exclusive access to data and therefore may either abort themselves or force other transactions to abort on conflict. An inefficient design or implementation may cause excessive or prolonged conflicts which directly harm performance.
The commit phase begins by attempting to acquire (lock) every ownership record guarding writes involved in the transaction. It does so by performing an atomic compare and exchange operation provided by the underlying DSM on the remote ownership record. It is worth mentioning again that the LSB of each ownership record is the lock bit, and that the LSB set to 1 indicates a locked ownership record while 0 indicates an unlocked ownership record. It is also worth mentioning that a transaction needs not explicitly abort when initially encountering a locked ownership record. The atomic compare and exchange operation uses the ownership record value stored in the writeset for each ownership record with the LSB set to 0 as expected value, and the same ownership record value with the LSB set to 1 as desired value. Acquisition of the ownership record is successful if and only if the atomic compare and exchange operation is successful, which means that the stored and remote ownership record version numbers are equal and that no other transaction currently holds the ownership record for writing.

If all ownership records in the writeset are successfully acquired, the data copies stored in the writeset containing changes made by the transaction are written to DSM memory followed by a blocking DSM release operation ensuring that all changes are propagated to their respective home nodes. Finally, each ownership record is unlocked through an atomic compare and exchange operation incrementing the version counter by one and setting the LSB to 0.

If any ownership record in the writeset can not be acquired, all ownership records are unlocked without incrementing the version counter and the transaction is aborted. Algorithm 2 details the implementation of the commit phase.

### 3.4.3 Aborting and restarting transactions

When a transaction decides to abort due to a transactional conflict, it is important to restart the entire transaction without carrying any information from the previous execution forward. This is important because we abort at the first conflict, and we do not know about any future conflicts that may occur during the restart. Therefore, all metadata collected by the transaction is discarded by completely resetting the read- and writeset. Additionally, the DSM data involved in the conflict can no longer be considered coherent, so it must be invalidated. Once this is done, the transactional memory system proceeds to restart the transactional code block again.

### 3.5 Initial design overview

In the previous sections of this chapter a fully functional software implementation of transactional memory has been described. The system utilizes ownership records stored in DSM memory space to govern concurrency and coherence. One thread can only execute one transaction at a given time, and transactional metadata is thread specific. Many threads may however execute on one DSM node concurrently. This system fulfills the atomicity, isolation and consistency requirements imposed on transactional memory, and accomplishes this without the use of software messengers by enforcing that each transaction decides on its own whether to commit or abort, and by utilizing RDMA communication.
success = true;

**LOCKING PHASE**

```plaintext
default writeset entry do
  foreach index-orec pair do
    if success == true then
      Attempt to lock ownership record remotely;
      if locking fails then
        success = false;
      end
    end
  end
end
```

**WRITEBACK PHASE**

```plaintext
if success == true then
  Write buffered data to local node cache;
  Release data to remote home nodes;
end
```

**UNLOCKING PHASE**

```plaintext
if success == true then
  foreach locked ownership record do
    Unlock remote orec with incremented counter;
  end
else
  Invalidate DSM data;
  foreach locked ownership record do
    Unlock remote orec;
  end
end
```

return success;

**Algorithm 2:** The data commit phase of transaction commit.
facilitated by the underlying DSM system to read and write DSM data. An overview of the system is shown Figure 7.

However, while this system fulfills the basic requirements for transactional memory, it does not fulfill all design principles described in Section 3. Most importantly, it does not minimize the amount of remote communication nor does it distribute coherence. In fact, the coherence protocol employed so far is a major bottleneck in larger systems. As a result, it does not perform or scale very well, which can later be observed in Chapter 4. In the remaining sections of this chapter, the various design flaws will be explored along with a number of major optimizations to overcome them along with improving performance and scaling.

3.6 Initial design flaws

In a distributed environment, it is of utmost importance to minimize the amount of remote communication necessary in order to optimize performance. Remote communication is often orders of magnitude slower than local communication, and the system design summarized so far in Section 3.5 fails to adopt this principle.

A simple example highlighting this issue can be seen in Listing 2. If x[0] and y[0] reside on different DSM pages, two separate DSM fetch operations will be issued, one for each memory access, in order to bring the pages into the local DSM cache. Additionally, both the read and the write operation will trigger remote reads fetching the ownership record values corresponding to the DSM addresses of x[0] and y[0]. Finally, during validation a remote fetch operation will be issued to compare the stored and the remote value of the ownership record involved in the read operation, and two remote operations will be issued in order to acquire and commit each ownership record involved in the write operation. This results in a worst case scenario of seven remote operations for two transactional actions. Even in an optimistic case the number of remote operations will be high, as there exists no way of safely reusing locally cached data.

Therefore, multiple alterations have to be made in order to minimize the amount of remote operations performed by transactions. Additionally, for the optimistic approach and the design philosophy inherited from ArgoDSM discussed in Section 3 to be true, additional alterations and improvements are required.

In order to minimize remote communication and closer align with an optimistic philosophy, the concept of selective self-invalidation along with an own-
Incrementing a counter in transactional memory can potentially trigger seven remote operations even without encountering a transactional conflict.

3.7 Selective self-invalidation

Each time a transaction aborts, it is a natural consequence that at least some DSM data cached on the node of the executing transaction is no longer guaranteed to be coherent. Therefore, this data has to be invalidated so that subsequent read or write operations to it operate on coherent data. So far, this has been achieved through a DSM acquire operation, self-invalidating all data in the calling node’s DSM cache. Invalidating all data on a node ensures correctness but can carry a severe performance penalty and directly opposes the principle of trading bandwidth for latency as any spatial locality obtained by fetching more data than necessary is lost each time a transaction aborts. By continuously invalidating the entire DSM cache on a node, other transactions executing on the node will be forced to carry out additional remote operations on data that may still have been coherent at the time of invalidation.

In order to avoid a larger than necessary performance penalty, selective self-invalidation (SSI) is implemented. This invalidation call takes a DSM address and a data size, self-invalidating all pages contained by the specified data region instead of all pages in the DSM cache. Not only does this have positive performance implications for other transactions executing on the same DSM node, it also facilitates further optimizations. The performance impact of SSI is shown in Chapter 4.3.

3.8 Ownership record cache

As mentioned in previous sections, transactional memory is inherently fit for read-heavy workloads that result in few conflicts, but where conflicts still have to be guarded against. It therefore makes sense for any optimizations to primarily target these types of workloads.

One can extend the notion of expecting few conflicts to include the expectation that ownership records accessed by multiple transactions are rarely, or never, used for writing. This is a natural consequence of the fact that only write transactions can induce conflicts. Therefore, it is wasteful to read ownership records remotely time after time if subsequent accesses could be made locally under the assumption that the ownership record has not been written remotely. A node-wide ownership record cache is introduced to facilitate this.

Each DSM node hosts one ownership record cache capable of fully replicating the ownership record structure. All transactions executing on the node collaboratively read and write data from the local ownership record cache. This approach allows each transaction to be served cooperatively by the cache, reducing the performance impact of conflict detection and resolution.
orate in updating this ownership record cache, so that if one thread remotely fetches an ownership record during a transaction it also inserts it into the ownership record cache. Other transactions executing on the same node may use this cached value instead of remotely fetching it again while performing transactional read or write instructions. Additionally, this allows for exploitation of spatial locality. As ownership records are stored sequentially in memory and each DSM fetch reads at least one DSM page, adjacent ownership records are also entered into the ownership record cache, emulating the effect of ownership record prefetching.

The ownership record is represented in memory by an array of structs containing ownership record values and associated metadata, including a mutex guarding access to the struct. Each struct is padded to a number of hardware cache lines to avoid false sharing on mutex operations. A detailed view of one cache struct can be seen in Figure 8.

By storing accessed ownership records in the ownership record cache, not only is the number of remote operations reduced in the optimistic case but reuse of cached DSM data is also made possible. Here, one has to use caution as DSM data and ownership records are not necessarily coherent at the same time due to the fact that an ownership record may guard more than one DSM address. Table 1 illustrates the four possible combinations of up to date and out of date ownership records and DSM data. If a transaction fails validation or commits due to an outdated ownership record, one cannot rely on the DSM data cached on the node as it may have been written since. This is trivial and DSM data has to be invalidated before use again (case 3 and 4). However, it is not trivial to decide whether DSM data is up to date when cached ownership records are up to date (case 1 and specifically 2).

The status flag and address list of each ownership record cache entry enables the transactional memory system to decide whether cached DSM data may safely be used or if it has to be fetched remotely again. The status flag has three states: valid, valid_inactive and invalid. The address list stores all DSM

<table>
<thead>
<tr>
<th>orec</th>
<th>data</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>up to date</td>
<td>up to date</td>
<td>no action necessary</td>
</tr>
<tr>
<td>up to date</td>
<td>outdated</td>
<td>dangerous, must prevent</td>
</tr>
<tr>
<td>outdated</td>
<td>up to date</td>
<td>no action necessary</td>
</tr>
<tr>
<td>outdated</td>
<td>outdated</td>
<td>no action necessary</td>
</tr>
</tbody>
</table>
addresses accessed through this ownership record.

If an ownership record is \textit{valid-active} and the DSM page address through which it is accessed exists in its address list, cached DSM data on the node is at least up to date with the ownership record and is safe to use. If the DSM address does not exist in its address list, only the DSM page containing the address is invalidated through SSI, and the address is then added to the address list for the cached ownership record.

If an ownership record instead is \textit{invalid}, it has either not been touched yet or it has been invalidated by another transaction. The ownership record value must be fetched by a remote DSM read and entered into the cache, and the calling DSM address must be invalidated through SSI to ensure that it is at least up to date with the fetched ownership record. Finally, the status flag may be set to \textit{valid-active} and the calling address added to the address list.

The \textit{valid-inactive} status flag can be considered a special case of \textit{invalid} where the ownership record value has been prefetched but not yet used. Therefore, the DSM data it guards cannot be guaranteed to be coherent and must be invalidated. Additionally, any addresses that have accessed the ownership record in its previous state are stored in the address list, and must be invalidated through SSI. The ownership record status flag may then be set to \textit{valid-active} and the calling address logged in the now empty address list.

Delaying invalidation of DSM data when an ownership record is prefetched into the ownership record cache, as opposed to aggressively invalidating pages upon prefetch, is a decision made consciously to reduce the number of unnecessary validations in the case where a prefetched ownership record is not used before a newer version is prefetched again. It is counterproductive to self-invalidate DSM data that other transactions may access before it is absolutely necessary to do so.

### 3.9 Distributing coherence

The final optimization necessary in order to better align with the ArgoDSM design philosophy is to distribute coherence. The underlying DSM system by default stores all DSM data on the first DSM node (Node 0) while there is sufficient memory available, and only when more memory than what is available on the first node is needed does it continue to subsequent nodes. While storing all transactional metadata on one DSM node achieves spatial locality, it is not scalable when the number of DSM nodes increases. At some point, the number of concurrent requests to the node where metadata is located will exceed both hardware and software capabilities and subsequently create a bottleneck for the whole system. In order to prevent this, \textit{stripe allocation} is introduced. Stripe allocation allows for data to be allocated contiguously in the DSM address space, while physically distributed in stripes across all DSM nodes. Specifically, DSM data is allocated in stripes of 16 kB in a round-robin fashion across all participating DSM nodes. This ensures that, for a typical distributed application using sufficient memory, the allocated memory and the ownership records guarding it is distributed fairly across all DSM nodes, reducing the risk of bottlenecks due to communication congestion on one node. The performance impact of stripe allocation is reported in Section 4.3.
3.10 Final design overview

A final design can now be presented and compared to the prototype design. The introduction of an ownership record cache and selective self-invalidation means that the algorithms for validation and commit need to be altered. Finally, the example given in Section 3.6 is revisited and explored with the optimized design in mind.

3.10.1 Ownership record cache

Figure 9 illustrates the addition of the ownership record cache, which is the only visible change compared to the design depicted in Figure 7. The obvious impact of the ownership record cache is that transactions no longer access ownership record values remotely during read and write operations, but instead query the ownership record cache for values. The ownership record cache returns the ownership record values requested, which are either already present in the cache or fetched remotely on cache miss, as well as ensuring that reading DSM data from the calling address is safe.

3.10.2 Read validation

Algorithm 1 presented in Section 3.4.1 will need minor changes to accommodate the addition of the ownership record cache. The necessary changes are highlighted in Algorithm 3, green lines representing additions and red lines representing removals. The optimized design utilizes SSI to invalidate only the DSM pages necessary to fetch up to date ownership record values, while the prototype design invalidated the whole DSM cache. This ensures that validation of one transaction does not interfere more than necessary with other non-conflicting transactions executing on the same node. Finally, an ownership record that fails validation must be invalidated in the ownership record cache so that subsequent transactions using the ownership record do so with an up to date value.

3.10.3 Write commit

Much like for read validation, the write commit Algorithm 2 presented in Section 3.4.2 needs minor alterations. The revised algorithm accommodating the ownership record cache can be seen in Algorithm 4. If locking an ownership record fails, the ownership record must be invalidated in the ownership record cache.
success = true;
Invalidate DSM data;

foreach readset entry do
  if success == false then
    Invalidate DSM data;
    break;
  end

foreach index orec pair do
  SSI DSM page if not done yet;
  if local orec != remote orec then
    if success == true then
      success = false;
      break;
    end
    Invalidate entry in orec cache;
  end
end

return success;

Algorithm 3: The read validation phase of optimized transaction commit.

cache for the same reason as invalidation is necessary after failed read validation. If write commit is successful, an attempt to update the ownership record cache is also made. This attempt is not blocking and instead uses a trylock, as any ownership record that is currently locked has already been updated or may be invalidated by a transaction currently in progress.

3.10.4 Optimized example

In Section 3.6, the worst case for the example given in Listing 2 was shown to be seven remote operations. During execution of the transactional code block up to four remote operations were required, two to read DSM data and two to read ownership records. During validation and commit, up to three operations were required in order to validate and commit the transaction. The optimizations made can unfortunately not reduce the number of operations in the worst case, but may significantly improve the optimistic case. If both ownership records and DSM data are successfully cached on the calling DSM node, no remote operations are required during transaction execution. Three remote operations are still necessary in order to validate and commit the transaction. The optimistic case has been reduced from seven to three remote operations, while the worst case still remains expensive in terms of remote communication. Chapter 4 explores the performance impact of each optimization.
success = true;

**LOCKING PHASE**

```
foreach writeset entry do
    foreach index-orec pair do
        if success == true then
            Attempt to lock ownership record remotely;
            if locking fails then
                success = false;
                Invalidate orec cache entry;
            end
        else
            Invalidate orec cache entry;
        end
    end
end
```

**WRITEBACK PHASE**

```
if success == true then
    Write buffered data to local node cache;
    Release data to remote home nodes;
end
```

**UNLOCKING PHASE**

```
if success == true then
    foreach locked ownership record do
        Unlock remote orec with incremented counter;
        Attempt to add new value to orec cache;
    end
else
    Invalidate DSM data;
    foreach locked ownership record do
        Unlock remote orec;
    end
end
```

return success;

**Algorithm 4:** The data commit phase of transaction commit.
4 Evaluation

The transactional memory system has been evaluated using a synthetic random access algorithm on the Tintin Cluster (now decommissioned) at Uppsala University. Transactional memory is compared to cohort locking [19] implementing the same algorithm with various different lock granularities. Furthermore, the impact of each major optimization in the transactional memory system is evaluated.

4.1 Evaluation environment

The Tintin Cluster at Uppsala University provides 160 Linux equipped compute nodes interconnected with QDR Infiniband. Each compute node is equipped with dual AMD Opteron 6220 “Bulldozer” running at 3.0 GHz and 64 GB of RAM (16 nodes carry 128 GB RAM). In total, the Tintin Cluster provides access to 2560 cores. For each evaluation, between one and 32 nodes running up to eight threads per node were used.

4.2 Evaluation algorithm

A synthetic random access algorithm was developed in order to illustrate performance at varying levels of read- and write load, as well as being able to illustrate the contribution of each major optimization in the transactional memory system. The algorithm operates on an array of 65536 integers contiguously allocated through the underlying DSM system, and is evaluated using between one and 256 cores running transactions in parallel for 30 seconds, after which the total number of transactions completed is used as measurement of performance. The relatively small data structure was chosen to ensure conflicts appear without enforcing long-living or large transactions.

A transaction in the synthetic random access algorithm performs three transactional operations along with simulating 500 µs local work through sleeping. The three operations are randomly (weighted by desired write load) chosen to be either three read operations or two read operations and one write operation, where the former is considered to be a read transaction and the later a write transaction. Finally, the same algorithm has been adapted to use cohort locks instead of transactions, with three different lock granularities: one global lock
while(1){
    // if runtime has passed, exit
    if(elapsed >= runtime){
        break;
    }
    // pick random elements
    writeflag = (write(rng) < writepercent) ? true : false;
    x = element(rng);
    y = element(rng);
    z = element(rng);
    // run transaction
    atomic_tx([&]()
    {
        if(writeflag){
            val = read_tx(&iarr[x]);
            val += read_tx(&iarr[y]);
            write_tx(&iarr[z], tid);
        }else{
            val = read_tx(&iarr[x]);
            val += read_tx(&iarr[y]);
            val += read_tx(&iarr[z]);
        }
        // simulate local work
        std::this_thread::sleep_for(std::chrono::microseconds(delay));
    }); // end transaction
} // end while

Listing 3: Evaluation code running on each thread in the system for a specified time. For clarity, code initializing variables and fetching thread data is omitted.

for the entire array (global), one lock per 1024 elements (coarse-grained) and one lock per 16 elements (fine-grained).

The code running transactions in each thread is shown in Listing 3. To ensure consistency, all random elements of the algorithm are chosen outside of the transaction. Elements and writeflags are chosen using std::mt19937, the standard implementation of the Mersenne Twister pseudorandom number generator [20] seeded with std::random_device. Depending on whether a read or write transaction is made, a different fork is executed before sleeping to simulate local work. If the transaction is completed, a new iteration begins unless the maximum time limit has been reached, thus ensuring a maximum of one transaction completing past the time limit. This is in the grand scheme irrelevant as the number of total transactions completed for each simulation was typically in the regions of $10^4$ to $10^7$. The cohort equivalent essentially replaces each transaction with one lock per read and one per write operation, with the exception of the global lock which is only taken once and held across all three operations.

4.3 Results

In order to determine scaling across a varying number of nodes and threads, along with scaling for varying read- and write load, a number of simulations were performed. Four differently optimized implementations of the transac-
tional memory system were compared against each other and three cohort lock implementations of different lock granularity as described in Section 4.2. The different versions of the transactional memory system were basic transactional memory without optimizations, transactional memory with SSI, transactional memory with SSI and ownership record caching as well as fully optimized transactional memory with SSI, ownership record caching and stripe data allocation.

Additionally, each comparison was made for nine different read- and write loads ranging from 0% write transactions to 25% write transactions. It is worth remembering that a write transaction consists of two read operations and one write operation, as opposed to three write transactions. This choice was made to closely reflect a real implementation, where writing data to one location is commonly preceded by reading data from another location. Finally, in order to better illustrate scaling for different implementations, both linear and logarithmic versions of the same data are presented.
4.3.1 Low write load

![Graph 1](image1)

**Figure 10:** Results of simulation under low write load on linear scale.

![Graph 2](image2)

**Figure 11:** Results of simulation under low write load on logarithmic scale.
4.3.2 Medium write load

![Graph showing results of simulation under medium write load on linear scale.](image1)

**Figure 12:** Results of simulation under medium write load on linear scale.

![Graph showing results of simulation under medium write load on logarithmic scale.](image2)

**Figure 13:** Results of simulation under medium write load on logarithmic scale.
4.3.3 High write load

Figure 14: Results of simulation under high write load on linear scale.

Figure 15: Results of simulation under high write load on logarithmic scale.
Transactional memory with no conflicts can be expected to scale more or less linearly, which can indeed be observed in Figures 10 and 11. As write load increases, so do conflicts, and as a result of this transactional memory scaling decreases across the board which is shown in Figures 12–15. However, regardless of the write load, scaling appears to continue to 256 cores. The case for 512 cores and above was not investigated due to cluster availability and queue times, but one can make an educated guess by inspecting the figures that scaling continues past 256 nodes in the case of fully optimized transactional memory. Cohort locks perform significantly worse than transactional memory for the task as the number of cores grows beyond 16, with fine grained lock granularity showing scaling at a similar rate to transactional memory once the number of cores increases, but with significantly worse throughput.

The impact of each transactional memory optimization can be observed in each of the Figures 10–15.

• Basic transactional memory implemented using atomic instructions and restricted to acquire and release functionality through the DSM performs well in comparison with cohort locks up to 128 cores, but suffers scaling issues beyond this point.

• Selective self-invalidation provides a significant improvement in performance over basic transactional memory restricted to node-wide release and acquire operations, but does not significantly improve scaling at and beyond 128 nodes.

• The ownership record cache, similarly to selective self-invalidation, provides a big performance improvement over basic transactional memory and transactional memory with selective self-invalidation, but neither optimization so far manages to deal with scaling past 128 nodes.

• Fully optimized transactional memory with stripe allocation alleviates the scaling issues at a low cost of slightly reduced throughput for a lower amount of cores (64 and below). This is likely due to the fact that transactional memory remote communication overhead creates a bottleneck for larger distributed systems if all data is allocated on the same node (including ownership records), and allocating it in a distributed fashion prevents bottlenecks at the cost of slightly decreasing data locality.
In this thesis, a scalable easy to use distributed software transactional memory extension of DSM has been presented and compared to alternative lock-based DSM synchronization. Transactional memory has been proven to be a worthy contender to locks in order to handle data races in software distributed shared memory, and significantly outperforms locking for the random access benchmark evaluated in Chapter 4. Furthermore, the importance of local decision making is shown by adding an ownership record cache and implementing selective self-invalidation, which allow transactions to optimistically execute based on locally cached data. This significantly improves performance by reducing the amount of remote communication necessary for the average transaction. Finally, the importance of distributed data allocation is shown by implementing stripe allocation allowing scaling past 128 compute cores by removing communication bottlenecks in the system.

5.1 Problems and limitations

Implementing transactional memory on top of software DSM brings both advantages and disadvantages. The fact that a shared cache exists on each DSM node allows transactional memory to access remote data locally, provided that coherence can be guaranteed. However, it also means that transactional memory has to adhere to DSM rules along with DSM implementation specifics.

The underlying DSM implementation relies on node-wide release and acquire to ensure coherence, creating problems if one wishes to invalidate a single page or even worse, a single cache line. To combat this, SSI was implemented. However, due to limitations imposed by the underlying software DSM, where the coherence protocol relies on a cache control granularity of one page, SSI is restricted to invalidating a single page at the very least. This can create unnecessary invalidations and subsequently additional remote operations, but will likely only become an apparent issue for applications utilizing very small amounts of memory. In order to work around this, instructing threads on the same node to work on different memory pages can be highly beneficial.

While transactional memory by definition prevents the user code from deadlocks, it does not guarantee livelock prevention. A livelock can be difficult to prevent or detect, as it is simply a special case of resource starvation in which no progress is made even though neither involved process stalls. In the case
of transactions this can appear when several transactions cause each other to abort repeatedly. However, in a distributed setting with long remote latencies repeated aborts in the same patterns are unlikely to happen, and therefore livelock prevention has been left as future work.

5.2 Future work

While this thesis prevents a fully functional software transactional memory implementation, several features and optimizations still remain to be implemented.

The current implementation does not prevent transactions with side effects, but neither does it support side effects, forcing users to delay side effects until after a transaction has completed. Any operations with side effects such as input or output are executed once per transaction iteration and, in order to prevent repeated or faulty side effects, any side effects should be delayed until after transaction commit ensuring that only a successfully committed transaction may perform actions carrying side effects. This was not seen as vital for the implementation and thus qualifies as future work.

Livelock prevention can be implemented through preventing a transaction from aborting and restarting after a certain number of aborts. However, if a number of transactions are forbidden to abort, the system needs to ensure that no deadlock can occur. By locking ownership records in an incremental fashion according to their index, one can make sure that one transaction can always make progress, but it is debatable whether immutable transactions will improve performance in any case. A solution that does not explicitly guarantee livelock prevention but greatly reduces the probability of livelocks is to apply a random or increasing back-off delay to restart upon transaction abort, such that livelocking transactions receive different delays and therefore eventually break livelock.

In the current transactional memory implementation, each ownership record protects four bytes of distributed shared memory, and this value along with the total amount of ownership records is defined in source code and can therefore not be altered at runtime. In practice, the optimal size of an ownership record and the amount of ownership records is application dependent, and therefore allowing external definitions with certain limitations may be beneficial. By using data from previous executions, the optimal ownership record structure could be determined through successive executions. A very low percentage of transaction aborts or a low percentage of write transactions may indicate that conflicts are unlikely, and therefore remote operations could be reduced either by reducing the amount of ownership records or by increasing the amount of distributed memory an ownership record protects. High abort rates or a high amount of write transactions may favour an increased amount of ownership records to reduce the amount of artificial conflicts created. However, further research on this is needed before any definitive conclusion can be drawn.

Transactions complete independently and pay no attention to other transactions in the system as long as no conflicts occur. For the sake of correctness this is beneficial, but whether it is optimal for performance or not is uncertain. For each commit, on top of atomically updating ownership records remotely, a node-wide release operation is performed propagating any data changes to their respective home nodes. If one was to group non-conflicting transactions on one
node together, without delaying their commit phase too much, one could reduce
the number of release calls and possibly improve performance. However, for al-
ready conflict-heavy workloads the delayed transaction commit may result in
creating additional conflicts, and therefore more research on this topic is neces-
sary in order to understand the impact and determine whether or not it would
be beneficial.

Finally, as the evaluation of the transactional memory system consists of
an entirely synthetic algorithm, a comparison to other distributed transactional
memory implementations by adapting a benchmark suite such as STAMP [21]
or LeeTM [22] to run on this implementation may be beneficial. Due to time
limitation, this was left as future work.


