COMPILER AND RUNTIME TECHNIQUES FOR SOFTWARE
TRANSACTIONAL MEMORY IN PARTITIONED GLOBAL ADDRESS SPACE
LANGUAGES AND RUNTIME LIBRARIES

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Srinivas Sridharan

________________________________________

Peter M. Kogge, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
November 2010
Abstract

by

Srinivas Sridharan

The Partitioned Global Address Space (PGAS) model has emerged as a promising abstraction for programming large-scale systems. In PGAS, threads share a common heap address space, which may in reality be partitioned and distributed across independent memories. Concurrency control mechanisms are thus required to ensure threads see each other’s updates in a consistent manner. The real challenge in building such mechanisms is that the memory locations that need to be updated in an atomic fashion may reside in different memory partitions. In this dissertation, we develop solutions based on Software-based Transactional Memory (STM) mechanisms to address the programmability and performance aspects of this challenging problem. STM mechanisms primarily guarantee that transactions, i.e. code sequences that access shared state, either execute as a single atomic operation or retry their operation in case such guarantees cannot be provided.

This dissertation makes the following contributions: First, we showcase the programmability benefits of providing language support for atomic transactions over lock-based approaches. Second, we develop first-of-its-kind compiler techniques for mapping these high-level language constructs to low-level STM designs. Third, we develop
STM runtime implementations that not only guarantee correctness for concurrently executing transactions, but does so in an efficient manner. For instance, we demonstrate for the first time how STM procedures can be implemented in a non-blocking manner, and use them to overlap computation and communication within a transaction. In general, we demonstrate the feasibility of these techniques by implementing them in Chapel, a parallel programming language being developed by Cray Inc. as part of DARPA’s HPCS program, and GASNet, a runtime library that implements the PGAS abstraction. Overall, we show that STM implementations can be scaled to hundreds (if not thousands) of nodes executing tens of thousands of threads, while exhibiting performance that exceeds, if not just matches, the performance of lock-based approaches.
Dedicated to,

*His Holiness Sri Sri Guruji Muralidhara Swamiji,*

*Malaipettu Lord Kalyana Srinivasa Perumal,* and

*Madhuri Sakhi sametha Sri Premika Varathan.*
CONTENTS

FIGURES ................................................................. vii

TABLES ................................................................. ix

ACKNOWLEDGMENTS ................................................ x

ABBREVIATIONS ...................................................... xiii

CHAPTER 1: INTRODUCTION ............................................. 1
  1.1 Motivation ....................................................... 1
  1.2 Software Transactional Memory for Global Address Space Applications 3
    1.2.1 Limitations of Lock-based Approaches .................... 3
    1.2.2 Shared–Memory Software Transactional Memory .......... 4
    1.2.3 Distributed–Memory Software Transactional Memory ...... 5
  1.3 Objectives ...................................................... 6
  1.4 Approach ....................................................... 6
  1.5 Contributions .................................................. 8
  1.6 Dissertation Roadmap ......................................... 9

CHAPTER 2: BACKGROUND WORK ....................................... 10
  2.1 Partitioned Global Address Space Model ....................... 10
    2.1.1 Concurrency Control for PGAS Models .................. 12
    2.1.2 PGAS Communication Libraries ........................... 13
  2.2 An Overview of the Chapel Language .......................... 14
    2.2.1 Support for Concurrency Control ......................... 15
    2.2.2 Motivating Examples ..................................... 16
      2.2.2.1 HPCC Random Access Benchmark ..................... 16
      2.2.2.2 Random Access 2 Kernel ............................. 22
    2.2.3 Chapel Compiler and Runtime Implementation ............ 24
  2.3 An Overview of Transactional Memory Concepts ................. 27
    2.3.1 Basic Concepts .......................................... 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4.3.2</td>
<td>Breakdown of Local STM operations</td>
<td>140</td>
</tr>
<tr>
<td>6.4.3.3</td>
<td>Breakdown of Remote STM operations</td>
<td>140</td>
</tr>
<tr>
<td>6.4.3.4</td>
<td>Read/Write Set Characteristics</td>
<td>144</td>
</tr>
<tr>
<td>6.5</td>
<td>Summary</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>CHAPTER 7: CONCLUSIONS AND FUTURE WORK</td>
<td>146</td>
</tr>
<tr>
<td>7.1</td>
<td>Conclusions</td>
<td>146</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Work</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>APPENDIX A: CHAPEL CODE: HPCC RA KERNEL</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
<td>153</td>
</tr>
</tbody>
</table>
FIGURES

2.1 HPCC RA Kernel: Using atomic statement .......................... 18
2.2 HPCC RA Kernel: Using synchronization variables ............... 22
2.3 RA2 Kernel: Using atomic–statement .................................. 23
2.4 RA2 Kernel: Using on–statement ......................................... 24
2.5 RA2 Kernel: Using sync–variables ....................................... 25
2.6 Chapel Compiler and Runtime System ................................. 26
3.1 HPCC RA Kernel: Using fully-local STM Library Procedures .... 43
3.2 RA2 Kernel: Using PGAS STM Library Procedures ................ 44
3.3 RA2 Kernel: Using PGAS RPC STM Library Procedures .......... 46
3.4 Synchronous vs Asynchronous Execution .............................. 47
3.5 RA2 Kernel: Using Non-Blocking PGAS STM Library Procedures . 50
3.6 RA2 Kernel: Using PGAS RPC STM Library Procedures .......... 51
4.1 List of GTM Procedures .................................................. 55
4.2 RA2 Kernel: Using GTM’s Transactional Load and Store Procedures . 60
4.3 RA2 Kernel: Using GTM’s Non-Blocking Transactional Load/Store Procedures. .................................................. 61
4.4 RA2 Kernel: Using GTM’s Non-Blocking Transactional RPC Procedures 62
4.5 Performance of Red–Black Tree Kernel ................................. 75
4.6 Performance of Priority Queue Kernel ................................. 78
4.7 Performance of RA2 Kernel .............................................. 80
5.1 Implementation-Independent STM Interface ......................... 96
5.2 Non-transactional and Transactional Get Operation ................. 97
5.3 HPCC RA Kernel: ATOMIC Version C code .......................... 98
6.1 STM Statistics ......................................................... 109
TABLES

4.1 GTM PROCEDURE CALL VARIATIONS .......................... 56
4.2 ORNL JAGUAR XT4 SYSTEM CONFIGURATION PARAMETERS 73
6.1 ORNL JAGUAR XT5 SYSTEM CONFIGURATION PARAMETERS 107
6.2 CHAPEL CONFIGURATION PARAMETERS ..................... 108
ACKNOWLEDGMENTS

I would like to thank my parents, Girija Sridharan and J. Sridharan, and my sister Anusha for their prayers, encouragement, and for always looking out for my well-being. You have been a great source of courage during all these years away from home. I would like to thank my guru, His Holiness Sri Sri Muralidhara Swamiji, for his spiritual guidance and unrelenting support through numerous challenges, and for always being there for me and my family. I would like to thank my fiancée Aishwarya for always believing in me and making the last six months leading to the completion of the dissertation the most enjoyable time I can ever remember.

I would like to thank Dr. Peter Kogge for being an excellent advisor, and for his unflagging patience and encouragement throughout my graduate school years. Dr. Kogge not only provided me the support and advice as and when I required it, but also gave me a lot of independence to establish my own style and identity. Special thanks to Dr. Kogge and his family for hosting me at various occasions and making me feel at home.

I would like to thank Dr. Jeff Vetter of Future Technologies Group (ORNL) for playing the role of mentor and collaborator, and for being on my dissertation committee. Thanks for the insightful guidance you provided through our collaboration. My three year long association with the FT Group, both as an intern and a visitor, has immensely contributed to my research progress.

I would like to thank Dr. Brad Chamberlain of Cray Inc. for playing the role of mentor and collaborator. Our collaboration was not only very rewarding and enjoy-
able, but has directly contributed to my dissertation research. Thanks for being easily approachable, and for taking time to provide practical advice and detailed feedback throughout our collaboration.

Overall, I would like to thank Dr. Kogge, Jeff, and Brad for helping me reach my research goals and for being great role models through their dedication to high–quality research.

I would also like to thank Dr. Jay Brockman, Dr. Douglas Thain for agreeing to serve on dissertation committee and providing valuable feedback over the years.

I wish to thank the entire Chapel project team for their excellent work and making it possible for me to build on their software platform. Thanks to Steve Deitz for helping me understand the internals of Chapel implementation and helping fill the gaps in my understanding.

Thanks to Philip Roth of Future Technologies Group of of Future Technologies Group (ORNL) for providing a great perspective from the “other side” and providing great advice on completing the dissertation in a timely fashion. I would also like to thank Kenneth Roche for his advice and engaging conversations during my first two stints at ORNL.

I would also like to recognize and thank all the staff of ND CSE department, ND Immigrations Service Office (ISO), and ORNL who have helped me navigate the myriad requirements and paperwork with ease over all these years. In particular, I would like to thank Joyce Yeats (ND CSE) for being aware of the challenges faced by graduate students and enquiring about my well–being. Thanks also for taking the initiative in finding an outside committee chair.

I would also like to thank Dr. Patrick Worley of ORNL for providing access to computing time on Jaguar XT4 and XT5 systems at ORNL. The work presented in
this dissertation used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

Thanks to Prof. Venkateswaran (a.k.a Waran) of Waran Research Foundation (India), for playing a vital role in mentoring me early on during my undergrad years and for inspiring me to pursue a doctorate degree.

Thanks to the Sri Narayanan Sathyamoorthy, Sri Ramanujam, Sri Murali, and many others of Namadwaar USA for providing a great service by organizing various satsangs on a regular basis.

Last but not the least, I would like to thank all my friends at ND and elsewhere over the years for the great times we shared.
ABBREVIATIONS

\[\begin{align*}
HPC & \quad \text{High Performance Computing} \\
GAS & \quad \text{Global Address Space} \\
GTM & \quad \text{Global Transactional Memory} \\
HPCS & \quad \text{High Productivity Computing Systems} \\
PGAS & \quad \text{Partitioned Global Address Space} \\
STM & \quad \text{Software Transactional Memory} \\
TM & \quad \text{Transactional Memory}
\end{align*}\]
CHAPTER 1

INTRODUCTION

In this dissertation, we develop scalable high-performance Software Transactional Memory (STM) mechanisms for the Partitioned Global Address Space (PGAS) programming model. In particular, we describe a first of its kind compiler and runtime support derived from these mechanisms, and demonstrate the benefits of language support for such STM designs.

1.1 Motivation

High-Performance Scientific Computing deals with harnessing the power of large-scale parallel computers to solve a number of hard problems that otherwise cannot be modeled with such ease and efficiency. Today’s large-scale systems, more commonly called commodity clusters, are built using commodity processor, memory, and network components, and have shown to scale well over 100k processor cores. As it stands, a vast majority of the applications targeting these systems use a sequential language such as C, C++, and/or Fortran to specify the computation on a per-node basis, and rely on the Message Passing Interface (MPI) library for expressing the necessary parallelism and communication across compute nodes. In recent years, the rapid increase in size and complexity of these systems, the increase in software complexity of applications targeting these systems, and the limitations of current software infrastructure to adapt
to changes in hardware, (e.g. *multicore* processors), have all spurred a considerable amount of related work in a new class of programming languages and models.

Of these, the *Partitioned Global Address Space* (PGAS) model has emerged as a promising abstraction for programming large-scale systems. The PGAS model is a subset of the *Global Address Space* model and additionally exposes the concept of *data locality* to the application. In GAS, the entire heap memory address space is treated in a flat-manner even when physically distributed across independent memories. Remote data is accessed via one-sided communication operations akin to local load/store operations. This enables threads on a one node to access a remote node’s memory directly. This eliminates the need for explicit coordination between the source and target nodes as required by MPI’s two-sided messaging model. Additionally, lack of implicit caching of remote data eliminates the need for cache-coherency protocols as in *cache coherent Non-Uniform Memory Architectures* (cc-NUMA) and *software Distributed Shared Memory* (SDSM) systems.

The *Partitioned Global Address Space* (PGAS) model is a subset of the GAS model and additionally exposes the concept of *data locality* to the application. Since data is in fact physically distributed across distinct nodes, exposing the location of the data as being local or remote helps optimize for performance. More recently, an extension of the PGAS model called the *Asynchronous Partitioned Global Address Space* (PGAS) model has been proposed to further expose the application to asynchronous remote tasks to further optimize for performance.

Today, a number of programming languages and libraries, such as *Co-Array Fortran* (CAF) [51], *Unified Parallel C* (UPC) [31], Global Arrays (GA) [50], Chapel [21], and X10 [18], expose these concepts to the programmer using high-level programming abstractions. In most cases, these systems themselves rely on low-level primitives
provided by portable GAS communication libraries such as *Global Address Space Networking* (GASNet) and *Aggregate Remote Memory Copy Interface* (ARMCI).

### 1.2 Software Transactional Memory for Global Address Space Applications

GAS models, similar to their shared-memory counterparts, require *concurrency control* mechanisms to coordinate and synchronize concurrent accesses to shared data. The problem lies in ensuring concurrent updates to shared data occurs in a consistent manner with respect to each other.

#### 1.2.1 Limitations of Lock-based Approaches

Currently, the most commonly used mechanism for concurrency control in these systems is *locks*. Locks protect concurrent accesses to shared data using the property of *mutual exclusion*. This property guarantees that only one thread executes the *critical sections* corresponding to a given a lock at any given time, essentially serializing the execution of the critical section. These critical sections are identified by the programmer by enclosing the code sequences that modify shared data with calls to acquire the lock before entering the critical section and releasing at the end of the critical section. For example, the UPC language provides *upc_lock* and *upc_unlock* operations to acquire and release lock variables.

Despite their many advantages, locks have a number of programmability and performance limitations. Locks do not compose well; the order in which locks are acquired and released affect the correctness of the program. Moreover, the programmer when identifying critical sections implicitly enforces the mapping between the lock and the data-structures protected by that lock. The choice of the optimal granularity, namely *coarse-grained* locks or *fine-grained* locks, cannot always be determined easily.
Coarse-grained locks are usually easy to implement but serialize concurrent accesses unnecessarily. On the other hand, fine-grained locks are harder to reason about and most data-structures don’t lend themselves to this kind of locking. Further, given the distributed nature of GAS programs, blocking on remote locks tends to be too expensive and fault-prone.

1.2.2 Shared–Memory Software Transactional Memory

In the last few years, a lot of attention has been focused on an promising alternative over lock-based mechanisms for concurrency control called Transactional Memory (TM) [36]. TM extends the idea of database transactions to provide a general concurrency control mechanism for protecting shared data. TM guarantees code sequences that access shared state, namely critical sections or transactions, appear to execute atomically with respect to each other or not execute at all.

TM belongs to the class of optimistic concurrency control mechanisms allowing transactions to execute concurrently but providing additional techniques necessary to deal with conflicting accesses. Conflicts arise when two or more transactions accesses the same data and at least one of these accesses happens to be a write. In such cases one or more of the conflicting transactions are aborted and are required to retry their entire execution. In case of no conflicts, transactions are allowed to commit in which case they make their changes visible to the rest of the system.

Much of today’s work on TM has focussed on developing new concurrency control mechanisms for mainstream programming languages and have chiefly focussed on workloads that fall within the memory and processing capabilities of a workstation-scale multi-core systems. In this regard, researchers have proposed numerous hardware, software, and hybrid implementations of TM. We deal with software-based implemen-
tations of TM (STM) [56] given their flexibility in experimenting with different implementation policies as well as the lack of hardware support for TM in today’s systems.

Early STM designs used a library API based approach [24, 30, 54, 56]. This approach invariably exposes the programmer to the low-level implementation details, including the need to explicitly mark the individual load and store operations that access the shared state within the transaction. More recent efforts have proposed language extensions that enable programmers to intuitively identify transactions while using the STM runtime library as a compilation target for implementing these high-level language constructs. In its most common form, the programmer identifies the transaction, for example using the `atomic` statement, and relies on the compiler to automatically instrument individual load and store operations. Atomic statements provide a simple to use interface and lends itself to code composition.

Even with its numerous benefits over locks, the reception of STM has been lukewarm. Locks tend to have a performance edge over STM given hardware support for cache-coherency in shared-memory multicore systems. In STM, the dominant performance overhead comes from maintaining both metadata for each transactional operation and metadata for the shared-data structures. Rigorous compiler optimizations along with novel algorithmic and implementation strategies have helped improve the overall performance [1, 14]. Nevertheless, the baseline overheads of individual operations is still too large for applications to benefit from STM-based approaches.

1.2.3 Distributed–Memory Software Transactional Memory

The scientific HPC domain targeted by Chapel, UPC, GA, etc. require that applications scale to thousands of nodes in order to benefit from large aggregate memory and processing power. As such, locks are still the status quo in these libraries
and programming languages. Not surprisingly, researchers have experimented with Distributed-Memory STM designs that extend the transactional programming model beyond shared-memory multi-core systems [9, 20, 37, 41, 44].

In large-scale parallel systems, the cost of single-node STM metadata management is masked by the cost of intra-node communication [9]. Thus, distributed STM schemes perform better (or similar to) locks while making it easier to write data-race free parallel programs. However as we shall see in Section 2.3.4, current distributed STM designs either do not focus on GAS models [20, 37, 41, 44] or do not adequately fulfill the requirements of PGAS languages and libraries [9].

1.3 Objectives

We believe concurrency control mechanisms using STM may overcome the limitations of locks and provide additional productivity benefits to large-scale applications written in PGAS languages and libraries. In short, the primary objectives of this dissertation is to:

- Design and implement high-performance STM runtime libraries for PGAS systems that scale to hundreds (if not thousands) of nodes while matching, if not exceeding, the performance of lock-based approaches.
- Develop compiler techniques that can target these STM runtimes.
- Demonstrate the programmability benefits of language support for atomic transactions using representative benchmark codes and application programs in scientific HPC domain.

1.4 Approach

We took a three-step approach to addressing the above objectives. As a first step, we identify a comprehensive set of design principles for developing scalable high-
performance STM designs targeting GAS models. Among other things, we identify
the synchronous blocking abstraction of today’s STM library interfaces as a major
roadblock to building high-performance STM designs, especially on large-scale par-
allel systems. Thus, inspired by asynchronous non-blocking communication opera-
tions supported by most message-passing and one-sided communication libraries, we
propose the novel concept of asynchronous transactional operations and demonstrate
how these can be implemented using non-blocking STM abstractions (Section 3.3). In
particular, we show how communication latency can be efficiently tolerated by hav-
ing simultaneous data/metadata requests in-flight and allowing each of these to execute
asynchronously with respect to the process/thread issuing these requests.

Next, we design and implement Global Transactional Memory (GTM), a scalable
STM runtime library for PGAS runtime platforms. Given its library interface, GTM is
not intended to be used as end-users writing application code. Rather, GTM is intended
as a proof-of-concept design to showcase the opportunities and challenges involved in
implementing scalable STM libraries on large-scale systems. Most importantly, GTM
is the first STM design to support non-blocking STM procedures. We leverage this
capability to efficiently tolerate the long latency of remote communication events by
having multiple requests executing across the system and overlapping them with other
computation or remote communication work.

Our prototype implementation uses the GASNet communication library [11] and is
intended to be used in conjunction with GASNet to provide a rich set of blocking/non-
blocking transactional/non-transactional one-sided and remote procedure call opera-
tions. We developed three workloads, namely Red-Black Tree, Priority Queue, and
Bank Transaction, to evaluate the scalability and performance aspects of our GTM pro-
totype. As such, given its library interface,
Finally, we explore the issues involved in implementing compiler and runtime support for STM in PGAS languages. For the purposes of this study, we chose Chapel [15, 16, 21, 22], a general-purpose parallel programming language being developed by Cray Inc. as part of DARPA’s High Productivity Computing Systems (HPCS) program [25]. As a matter of fact, Chapel is one of the first languages to propose the atomic statement for specifying transactions that require distributed STM capabilities and thus provides an excellent vehicle for exploring the productivity benefits of STM for large-scale parallel applications. However, no prior effort has been undertaken to address the language, compiler, and runtime challenges in implementing these ideas. Our work is the first attempt to not only explore possible language semantics but also design and implement Chapel’s atomic statement using STM techniques.

Our work in Chapel involves active collaborations with the Chapel language designers. We are working towards the goal of making our enhancements to the Chapel compiler and runtime be included into the main Chapel source distribution and eventually be made part of future software releases to the Chapel user community.

1.5 Contributions

This dissertation explores the challenges in developing scalable STM solutions for PGAS languages and runtime libraries. To summarize, this dissertation makes the following key contributions:

- We provide guidelines for designing scalable STM runtime libraries for PGAS languages and libraries.
- We demonstrate that STM implementations can be scaled to hundreds (if not thousands) of nodes executing tens of thousands of threads, far beyond what has been demonstrated by any other distributed STM implementations.
- We propose the novel concept of asynchronous transactional operations and demonstrates how these can be implemented using non-blocking STM procedures. We
demonstrate the use of non-blocking STM procedures to overlap communication and computation within a transaction and tolerate remote communication latency.

- We identify and address various challenges in designing compiler techniques for atomic transactions in PGAS languages.
- We demonstrate the programmability benefits of PGAS language support for atomic transactions, especially over lock-based approaches.
- We identify various overheads that hamper performance and scalability, and suggest possible optimizations to overcome these limitations.

1.6 Dissertation Roadmap

This chapter describes the motivations, objectives, approaches, and overall contributions of the work presented in this dissertation. Chapter 2 describes the background work on PGAS systems, STM, and the Chapel language relevant to understanding the work presented in this dissertation. Chapter 3 overviews some of the design requirements and principles of PGAS STM designs and describes how asynchronous transactions can help tolerate communication latency in large-scale systems. Chapter 4 describes the design and implementation of GTM and describes the experimental results for the scalability and performance aspects of our GTM prototype. Chapter 5 describes the syntax and semantic issues surrounding Chapel’s atomic statement and the changes required to implement these semantics with the Chapel compiler and runtime system. Chapter 6 evaluates our implementation of atomic transactions in Chapel. We consider the programmability and performance benefits of atomic transactions, and analyze how it compares against lock-based versions of the same kernels. Finally, Chapter 7 summarizes the work presented in this dissertation and presents some opportunities for future work.
CHAPTER 2

BACKGROUND WORK

In this chapter, we present some background work relevant to the rest of the dissertation. This chapter is organized as follows: first we present the basic concepts behind the PGAS languages and runtime libraries in Section 2.1. Next we present a brief overview of the Chapel HPCS language and describe some of the internal implementation details of the Chapel compiler and runtime system in Section 2.2. Finally Section 2.3 present a brief overview of TM concepts and implementation techniques that can be found in today’s literature.

2.1 Partitioned Global Address Space Model

According to Culler and Gupta [23], a parallel programming model “specifies how parts of the program running in parallel communicate information to one another and what synchronization operations are available to coordinate their activities”. Such a model is usually embodied in a parallel language or runtime library, and maps generic language constructs to specific primitives provided by the underlying hardware or system software. Two of the most widely used parallel programming models are the message passing model and the global address space model. The message passing model provides each task (usually called processes) with private memory and uses explicit messages for communication between these entities. Synchronization is explicit as part
of the communication operation. In contrast, *global address space* paradigm makes
some part of the memory accessible to all processing entities (threads). Communication
occurs implicitly using read and write operations to memory. Further, GAS treats
the entire heap memory address space in a flat-manner even when physically distributed
across independent memories.

A third option, the *Partitioned Global Address Space* (PGAS) model has emerged
as a promising alternative to programming large-scale systems. The PGAS model is a
subset of the *Global Address Space* model and additionally exposes the concept of *data
locality* to the application. PGAS combines the benefits of both *message passing* and
*global address space* programming models.

Remote data is accessed via *one-sided communication* operations akin to local load-
/store operations. This enables threads on a one node to access a remote node’s memory
directly. Since data is in fact physically distributed across distinct nodes, exposing the
location of the data as being local or remote helps optimize for performance. This eliminates
the need for explicit coordination between the source and target nodes as required
by MPI's two-sided messaging model. Additionally, lack of implicit caching of remote data eliminates the need for cache-coherency protocols as required by other GAS implementations *cache coherent Non-Uniform Memory Architectures* (cc-NUMA) and *software Distributed Shared Memory* (SDSM) systems.

Today, a number of programming languages and libraries, such as *Co-Array Fortran* (CAF) [51], *Unified Parallel C* (UPC) [31], *Global Arrays* (GA) [50], *Chapel* [21], and X10 [18], expose these concepts to the programmer using high-level programming abstractions. In most cases, these systems themselves rely on low-level primitives provided by portable GAS communication libraries such as *Global Address Space Networking* (GASNet) and *Aggregate Remote Memory Copy Interface* (ARMCI).
2.1.1 Concurrency Control for PGAS Models

GAS models, similar to their shared-memory counterparts, require concurrency control mechanisms to coordinate and synchronize concurrent accesses to shared data. The problem lies in ensuring concurrent updates to shared data occurs in a consistent manner with respect to each other.

Currently, the most commonly used mechanism for concurrency control in these systems relies on locking the data in a manner that allows only one thread to access it at a time. We shall see specific reasons why these lock–based approaches fall short of addressing both programmability and performance requirements for writing large-scale parallel applications. Further, correctness too becomes a real challenge if we consider code composability with locks.

As a matter of fact, language support for concurrency control is one of the main innovations in many of the recent PGAS programming languages. For example, all the three languages originally initiated by the HPCS program have introduced new interfaces for specifying the required synchronization operations.

Chapel is one of the three languages initiated by the HPCS program, the other two being IBM’s X10 [18] and Oracle/Sun’s Fortress [4]. Since their inception all these languages listed the atomic construct as part of their language specification, albeit with widely varying usage and semantics.

X10’s atomic block is expected to be implemented using lock–free and wait-free synchronization mechanisms and not STM. Further, X10 requires all the memory locations accessed in the atomic block to be local to the locale (place in X10 terminology) on which the task (activity) is executing.

Fortress, unlike X10, relies on STM mechanisms to implement its atomic block. However, it is similar to X10 in not allowing remote operations within its atomic block.
Fortress uses the DSTM2 [39], a multi-core STM library, to implement its atomic construct. DSTM2 supports flat-nesting semantics for nested transactions, simple backoff based contention management policy for resolving conflicts, and obstruction freedom policy for guaranteeing forward progress. In addition to the atomic block, Fortress supports the 
\texttt{tryatomic} that supports conditional execution of the transaction only if the specified condition is satisfied. In Section 2.2, we shall look at the Chapel language in particular and see why lock based mechanisms are insufficient for programming PGAS systems.

2.1.2 PGAS Communication Libraries

PGAS languages and runtime libraries rely on underlying GAS communication libraries to implement the basic features for supporting the GAS abstraction. ARMCi and GASnet are two of the more popular GAS communication libraries.

Both these libraries rely on the \textit{Single Program Multiple Data} (SPMD) model as the primary execution model for expressing parallelism across multiple nodes. In this regard, a global memory location is identified using a pair of arguments: \textit{rank} or process identifier that uniquely identifies a specific target process and a virtual memory address within that process’s address space. This ordered pair logically represents a global pointer for a given address within the memory partition of the specified process and is valid throughout the system.

\textit{GASNet} [11] is a low-level networking library that provides a wide range of communication primitives for implementing parallel PGAS languages such as CAF, Chapel, Titanium, UPC, and X10. GASNet’s interface is partitioned into two layers. GASNet’s \textit{Core API} is based on \textit{Active Messages} (AM) [58] and is implemented specifically on each network architecture.
On the other hand, the *Extended API* provides high-level functionality for performing collective operations or one-sided *Remote Memory Access* (RMA) operations. GASNet’s extended API is implemented in a native manner in platforms that support one-sided operations and is implemented using the core API on all other platforms. GASNet provides three different threading environments: fully threaded and thread-safe, fully threaded but non-threadsafe, and single-threaded. While the first configuration allows multiple threads to invoke GASNet calls in a threadsafe manner, the second one allows only one thread to invoke GASNet calls at any given time. By *thread-safety* we mean the procedure or function supports multiple threads of control to invoke it at the same time.

*ARMCI* [49] is similar to GASNet in its functionality. It provides one-sided RMA operations that supports contiguous and noncontiguous (strided, scatter/gather, I/O vector) data transfers, active messages, and atomic operations. It has been used to implement GA and CAF. Unlike GASNet, ARMCI currently does not support a fully threaded thread-safe environment.

2.2 An Overview of the Chapel Language

Chapel [15, 16, 21, 22] is a general-purpose parallel programming language being developed by Cray Inc. as part of DARPA’s *High Productivity Computing Systems* (HPCS) program. Chapel is chiefly motivated by the growing inefficiencies and challenges in programming large-scale parallel systems. Chapel’s *global-view* abstraction for expressing distributed data-structures, and ability to expresses arbitrary parallel algorithms, generalized multithreaded execution model lends itself to developing code across a wide-range of parallel architectures – from shared memory multicore systems to distributed memory clusters.
The rest of this section is divided into three parts. Section 2.2.1 describes language support for different concurrency control mechanisms in Chapel. Next, we use two sample programs to motivate the benefits of Chapel’s atomic block construct over lock-based approaches. We use these examples to explain a number of high-level Chapel concepts and revisit them throughout the course of this dissertation. Finally, we describe the implementation of the Chapel compiler and runtime system in Section 2.2.3. The following material is not intended to provide a comprehensive description the Chapel language and its rich feature set. Readers interested in a more in-depth understanding of the language are referred to the references listed earlier.

2.2.1 Support for Concurrency Control

Chapel belongs to the class of recent PGAS languages that provide better language abstractions for concurrency control. Currently, two of the most important mechanisms in Chapel’s toolbox are synchronization variables and atomic statements.

Chapel provides type system support for identifying synchronization variables using the `sync` keyword. Synchronization variables are special variables that provide an additional bit of metadata in conjunction with regular data values. The additional bit, called the `full/empty` (FE) bit, indicated if the data part is “full” or “empty”. Chapel provides synchronized load and store operations that may operate on the data only if the FE bit meets certain pre-conditions. A synchronized load operation blocks until the FE bit is “full” then sets it “empty”, and returns the contents of the data part. A synchronized store operation blocks until the FE bit is “empty” then stores its values in the data part, and sets the FE bit to “full”. As such, synchronization variables are best suited for producer-consumer style synchronization but can also be used around critical sections to provide mutual exclusion (similar to regular locks).
Not surprisingly, Chapel designers recognized the limitations of traditional lock-based concurrency control interfaces and included the \textit{atomic}–statement as part of the original language specification. Chapel defines the \textbf{atomic} block construct as a flexible concurrency control mechanism for \textit{atomically} updating one or more memory locations in an all-or-nothing manner. Atomic block statements appear to execute as a single unit with respect to each other or not execute at all. They modify the global state in a consistent manner and ensure intermediate updates are not made visible.

One of the primary benefits of the \textbf{atomic} block over lock–based mechanisms is that it lends itself to code composition. Atomic blocks can be composed such that an \textit{inner atomic} block is contained within an \textit{outer atomic} block. Using locks in similar situations must ensure the same lock is not acquired twice or that locks are acquired in a specific order so as to avoid deadlocks. Chapel designers envisioned the use of high-performance STMs to implement the atomic construct. Our work, presented in Chapters 5 and 6, represents the first of its kind attempt to address the language, compiler, and runtime challenges in implementing Chapel’s \textbf{atomic} block construct using PGAS STM techniques (Section 2.3.4).

2.2.2 Motivating Examples

In this section we demonstrate the limitations of lock–based approaches and motivate the need for better concurrency control mechanisms using two Chapel code examples. We also explain a number of high-level Chapel concepts using these examples.

2.2.2.1 HPCC Random Access Benchmark

The \textit{HPC Challenge} (HPCC) benchmark suite [26] is a collection of seven kernels, and is used to rank the world’s fastest supercomputers [57]. Chapel’s version of the
HPCC benchmarks has won the HPCC Awards [40] for the Most Elegant Implementation for past two years [17]. It is one of the most commonly cited code examples, showcasing the various productivity benefits of the Chapel language.

Each of the HPCC benchmark kernels is designed to test a specific feature of a large-scale system. Of these, the Random Access (RA) benchmark is designed to test the limits of memory system performance since the updates do not follow temporal and spatial locality characteristics suited for processor caches.

Very briefly, this kernel performs a random sequence of updates on a large distributed array/table and measures the rate of updates in terms of Giga Updates Per Second (GUPS).

The kernel is divided into two phases. In the first phase the RA kernel performs a random sequence of $NU$ updates on a $m$ element distributed table/array $T$. Each of these updates is an XOR operation. The benchmark specification requires these updates be done in parallel but does require any concurrency control mechanism to guarantee correctness. This implies, some of these updates may be lost due to the lack of proper concurrency control mechanism to ensure atomicity. We refer to this phase also as the unsynchronized or update step.

A second phase reapplies the same sequence of updates as the first phase. Given the commutativity and associativity property of XOR operations, reapplying the same set of updates must reset the value of the table entries to their initial values. We refer to this phase also as the verification step. Since the verification step is also done in parallel, the updates have to be performed in a thread-safe manner in order to avoid any additional inconsistencies. Following the updates, we can calculate the number of lost updates based on number of table entries that have a final value different from their initial value.
const TableDist = new dmap(new Block(boundingBox=[0..m-1])),
TableSpace: domain(1,uint(64)) dmapped TableDist=[0..m-1];
var T: [TableSpace] uint(64);
const n = log2(m);

const UDist = new dmap(new Block(boundingBox=[0..NU-1])),
Updates: domain(1,uint(64)) dmapped UDist=[0..NU-1];

def indexMask(r: randType): randType {
    return r >> (64 - n);
}

forall (u, r) in (Updates, RandomStream()) {
    const idx = indexMask(r);
    T(idx) ^= r;
}

forall (u, r) in (Updates, RandomStream()) {
    const idx = indexMask(r);
    atomic T(idx) ^= r;
}

Figure 2.1. HPCC RA Kernel: Using atomic statement
The code listing in Figure 2.1 implements the HPCC RA kernel. First let us consider what it takes to declare a distributed array in Chapel (line 1 to line 3). Here, $T$ is declared as an one-dimensional array with its elements of type $\text{uint}(64)$ (line 3). $\text{TableSpace}$ is a domain, and defines $T$’s index set. $\text{TableSpace}$ is declared as an one-dimensional domain and its index type is $\text{int}(64)$ (line 2). It is initialized to store the indices from $[0..m-1]$. Since the size of the array is defined by its set unique indices, this makes $T$ an array of size $m$. In addition to this, $\text{TableSpace}$ is declared as a mapped domain using the $\text{dmapped}$ keyword. Mapped domains are domains that have a domain map. Domain maps specify how domains (and thus in turn arrays) are distributed across locales. Before we describe domain maps in greater detail, it is important to understand the concept of locales in Chapel.

A Locale is a first-class language concept in Chapel and helps users specify and reason about about the placement of data and tasks on a target architecture in order to tune for locality. Chapel defines what constitutes a locale based on the target platform. For example, on a distributed memory cluster, a locale refers to the SMP or multicore processor node. This follows the reasoning that tasks executing on a given node pay a huge price to access remote memory on other nodes compared to accessing local memory on their own nodes.

Thus, domain maps define the mapping of indices to memory locations across locales. In the above example, we use a Block distribution to partition the indices $[0..m-1]$ into approximately equal sized blocks and distributes it across all the locales (line 1). To summarize, the above code snippet declares an one-dimensional array and distributes it across the system in approximately equal sized blocks.

Now, accessing this array is quite straight-forward and follows the same syntax as accessing say a non-distributed array. Given an index $\text{idx}$, the distributed array $T$
can be accessed as follows: $T(idx)^r = r$, irrespective of the physical location of $T(idx)$. This is in part made possible by Chapel’s implementation of domains and domain maps, and plays a crucial role in Chapel’s support for global-view abstraction for accessing distributed data-structures.

Now, let us consider the update loop of this kernel (lines 13–16). The `forall` keyword implies the loop can be executed in a parallel manner. Each iteration of this loop corresponds to a single update operation. Our parallel loop happens to use a parallel zippered iterator. In other words, the loop specifies an element-wise iteration over the domain `Updates` and iterator function `RandomStream`, and returns a pair of values for each iteration of the loop.

Domains support parallel iterators over their index sets. The `Updates` domain represents the set of all NU updates using the indices $[0..NU-1]$. It uses the domain map `UDist` to distribute these NU updates as evenly as possible across all the locales. Thus, the loop may be executed in a parallel manner across all the locales, with each locale executing its own chunk of the NU updates.

The `RandomStream` is an iterator function that returns a stream of random values. Chapel iterator functions are functions that can return multiple values. Iterator functions are similar to regular functions but uses the yield statement to return values. However unlike the return statement, the iterator continues its execution following the yield statement and can thus return additional values. The iterator eventually finishes when it encounters a return or when the last statement of the iterator function is executed.

In the above case, `Updates` is the leader iterator and `RandomStream` is the follower iterator. This implies the `RandomStream` iterator follows the leader iterator and yields values on the behest of the leader. Thus, for each iteration of the parallel
zipper iterator also produces a new random number $r$. We use only the top $n$ order bits to obtain the array index (line 12), where $n$ is calculated from the size of the array. Finally, we update the table entry corresponding to index $\text{id}_x$ (line 13).

One minor observation that we can make here is that we do not actually use $u$ anywhere in the loop and therefore can avoid explicitly naming it. Chapel allows us to replace $u$ with a blank space as a convenience.

As we increase the number of locales, the probability that $i_d$x refers to table entry on a remote locale also increases. We can use the Chapel $\text{on}$ construct to optimize for data locality by sending the computation to the locale that is responsible for the data:

$$\text{on } T(i_d)x \text{ do } T(i_d)x \map{} r;$$

As one can easily observe, operating on the array in parallel without proper synchronization may lead to data-races. As such the first phase allows some of the updates to be lost. However, the verification phase requires some form of concurrency control to ensure no updates are lost. As we mentioned earlier, Chapel provides language support for concurrency control for this very purpose.

Figure 2.1 demonstrates the use of $\text{atomic}$–statement to specify the required concurrency control in an intuitive manner. We refer to this version of the kernel as RA–ATOMIC. Among other things, the $\text{atomic}$ construct relies on the compiler to choose the best implementation and helps separate the semantics from the implementation. We shall see how the $\text{atomic}$-statement is implemented in the course of this dissertation.

We can implement the verification loop using $\text{sync}$–variables as well. The code snippet in Figure 2.2 declares the table $T$ as an array of synchronization variables and uses compiler support to generate the correct synchronized load and store operations. We refer to this version of the kernel as RA-SDA. The compiler maps the read operation of $T(i_d)x$ (line 10) to a synchronized load operation and returns the value of the
```
var T: [TableSpace] sync uint(64);

forall (p, r) in (Updates, RandomStream()) {
    const idx = indexMask(r);
    on T(idx) do T(idx).writeXF(T(idx).readXX() ^ r);
}
```

Figure 2.2. HPCC RA Kernel: Using synchronization variables

table entry corresponding to the index idx. Similarly, the write operation on line 10 is mapped to a synchronized store operation. It is worth noting that all operations defined on the table are now automatically synchronized. Accessing the table in an unsynchronized manner can only be achieved by using special routines defined for sync–variables, such as readXX() and writeXF(). The readXX() method reads the data part of the sync–variable and completely ignores the FE bit. On the other hand, the writeXF() method writes into the data part and always sets the state to full irrespective of the start state. This change is reflected on the update loop (lines 6).

2.2.2.2 Random Access 2 Kernel

The Random Access Two (RA2) kernel performs a sequence of $NU$ updates to two random elements of a $m$ element distributed table/array T. The RA2 is a synthetic kernel derived from RA. However, unlike RA kernel, RA2 updates two table entries for every update operation. The code listing in Figure 6.15 demonstrates the use of atomic blocks for implementing the updates in a thread-safe manner. We refer to this version of the kernel as RA2–ATOMIC. Similar to the RA kernel, we can use Chapel’s on
const TableDist = new dmap(new Block(boundingBox=[0..m-1])),
TableSpace: domain(1,uint(64)) dmapped TableDist=[0..m-1];

const UDist = new dmap(new Block(boundingBox=[0..NU-1])),
Updates: domain(1,uint(64)) dmapped UDist=[0..NU-1];

var T: [TableSpace] uint(64);

def ra2_core (r, s) {
    const idx1 = indexMask(r);
    const idx2 = indexMask(s);
    on T(idx1) do {
        T(idx1) ^= r;
        T(idx2) ^= s;
    }
}

forall ( , r, s) in (Updates, RandomStream(seed1),
RandomStream(seed2)) do
    ra2_core(r,s);

forall ( , r, s) in (Updates, RandomStream(seed1),
RandomStream(seed2)) do
    atomic ra2_core(r,s);

Figure 2.3. RA2 Kernel: Using atomic--statement

construct to optimize for data locality. In this case the second on clause occurs within a
transaction and must be executed with transactional semantics as shown in Figure 2.4.

The code-listing in Figure 2.5 demonstrates the use of Chapel’s sync variables to
implement the RA2 kernel. Here the table is declared as an array of sync variables.
Developing a lock--based version of this kernel is not as straightforward and has to take
into account the following issues:

• Enforce a global order in which locks are acquired. This is required to avoid
deadlocks. We acquire the locks in the increasing order of index.
def ra2_core (r, s) {
    const idx1 = indexMask(r);
    const idx2 = indexMask(s);
    on T(idx1) do {
        T(idx1) ^= r;
        on T(idx2) do T(idx2) ^= s;
    }
}

Figure 2.4. RA2 Kernel: Using on–statement

- Release the locks in the order in which they are acquired.
- Ensure the same lock is not acquired twice. This may happen if idx1 and idx2 is protected by the same lock (line 16).

It should be noted that with synchronization variables, the update loop must be implemented separately using the special readXX and writeXF methods defined for synchronization variables.

As mentioned earlier, Chapel does not currently support the atomic–statement. Our work is the first of its kind effort in addressing the language, compiler, and runtime challenges in implementing Chapel’s atomic block construct using PGAS STM techniques and is described in Chapter 5 and Chapter 6.

2.2.3 Chapel Compiler and Runtime Implementation

Figure 2.6 shows the entire Chapel compilation process and the various components of the Chapel compiler and runtime system. The first step in Chapel’s compilation process involves the Chapel-to-C compiler. As the name suggests, it is a source-to-source compiler and converts the input Chapel program into C language code. This component is implemented as a sequence of passes and starts with building an intermediate
representation (IR) of the input source code. This representation, also called Abstract Syntax Tree (AST), is modified by the subsequent compiler passes eventually resulting in the C language code.

The second step involves linking the generated C code with Chapel’s runtime library. The Chapel runtime library plays a very important role in implementing Chapel’s programming and execution model in a portable manner. Among other things, Chapel’s runtime library is responsible for bootstrapping the Chapel program across multiple locales, handling intra-node task/thread management, and implementing inter-locale communication capabilities.

At the highest level Chapel’s runtime exports a high-level implementation independent interface without tying it to a specific implementation. The Chapel-to-C compiler embeds calls to this high-level runtime interface in the C code it generates. The high-

---

```plaintext
1 var T: [TableSpace] sync uint(64);

3 def ra2_core (r, s) {
4   var x: uint(64);
5   on T(idx1) {
6     if (idx1 < idx2) {
7       x = T(idx1);
8       T(idx2) ^= s;
9       T(idx1) = x ^ r;
10      }
11     }
12   else if (idx1 > idx2) {
13     x = T(idx2);
14     T(idx1) ^= r;
15     T(idx2) = x ^ s;
16   }
17   }
18 }
```
The level interface is an important glue between the compiler and runtime implementation and helps keep the compiler independent of the actual runtime implementation.

One of the essential aspects of the Chapel system implementation lies in its own use of Chapel code to implement a number of language features. For example, Chapel implements its `sync` variables, arrays, tuples, ranges, and domains using Chapel classes. This strategy has been an essential part of the language development and has helped in creating a simpler and robust implementation of the language.

The implementation organizes this code into different modules based on their functionality. Chapel `internal` modules implement certain essential features such as arrays and sync variables and is automatically included as part of every Chapel user/client program. Chapel `standard` modules implement commonly used functionality such as sort, timers, and is included by the programmer as and when required. Finally, support for distributions is implemented by `dists` modules (not shown in Figure 2.6) and are also chosen based on user requirements.
2.3 An Overview of Transactional Memory Concepts

A transaction can be defined as a sequence of operations guaranteed to execute as an atomic unit, i.e. indivisible and instantaneous. Transactions were originally proposed in the context of databases. To ensure correctness, database transactions were required to satisfy the Atomicity, Consistency, Isolation, and Durability (ACID) properties. The atomicity property ensures transactions complete all of its constituent operations either fully or appear to have not executed at all. The atomicity property by itself is not sufficient to support atomic execution. The former is usually referred to as failure atomicity to differentiate it from the whole concept of atomic execution [42]. The consistency property ensures that transactions maintain all data-structure invariants. The isolation property ensures that transactions do not interfere with the results of other concurrently running transactions. Finally, the durability property ensures that committed results are never lost.

Transactional Memory (TM) [36] extends the ACI properties of database transactions to provide a general concurrency control mechanism for atomic update of shared data in a multithreaded environment. The durability property is usually ignored since threads share data via volatile memory unlike databases that use disks for storage. TM has been shown to provide performance advantages over locks by eliminating the unnecessary serialization and allowing concurrent access to disjoint data objects. From the programmer’s perspective, language support for TM mechanisms has shown to help develop composable code that is free from data-races. This section first describes the basic terminology of concurrency control mechanisms and TM (Section 2.3.1). Next we describe various design choices and policies for implementing TM in Section 2.3.2. These policies are independent of whether they are implemented in software (STM) [24, 30, 54, 56], in hardware (HTM) [36, 45], or a hybrid of both (HyTM) [46, 55].
In this dissertation we primarily focus on STM solutions given its flexibility in supporting them on today’s systems and given the lack of hardware based solutions in the foreseeable future. Overall, STMs are a better choice for research as they are more flexible for experimenting with different implementation policies.

Based on the target platform, we classify STM implementations into Shared Memory STM (SMSTM) and Distributed Memory STM (DMSTM). SMSTM designs primarily target shared-memory multicore processors, while DMSTM designs target distributed memory systems that extend TM beyond a single compute node. This includes STM designs for Distributed Shared Memory (DSM), PGAS, grid, and cloud platforms. We explore these in greater detail in Section 2.3.3 and Section 2.3.4 respectively.

2.3.1 Basic Concepts

Conflicts occur when concurrently executing threads access some shared data, and at least one of those accesses happens to be a write. If allowed to proceed these conflicting accesses may result in data races and leave the data in an inconsistent state. Conflict detection deals with detecting the occurrence of such conflicts, while conflict resolution deals with taking appropriate action to ensure correctness.

Pessimistic concurrency control mechanisms deal with conflict detection and resolution immediately following the conflict. This is usually implemented by mutual exclusion, where a thread acquires exclusive access to the object being modified using locks.

On the other hand, optimistic concurrency control mechanisms deal with conflict detection and resolution at a later point in execution. This type of control allows multiple threads to access object concurrently but additional mechanisms are necessary to ensure correctness.
Forward progress is a property that ensures that threads seeking to modify shared objects will eventually complete their operations. Pessimistic concurrency mechanisms provide a single thread exclusive access and blocks all other threads. These blocking synchronization techniques do not provide any guarantees for forward progress to the system as a whole when threads holding exclusive resources are either preempted or terminated. On the other hand, optimistic concurrency control implements nonblocking synchronization mechanisms that provide stronger forward progress guarantees. In other words, the implementation guarantees that one thread cannot stall all other threads in the system indefinitely.

Non-blocking synchronization supports three types of forward progress guarantees, namely: wait freedom, lock freedom and obstruction freedom. Wait freedom is the strongest guarantee where all threads contending for some shared resource make forward progress in a finite number of individual attempts. Lock freedom guarantees that at least one of the contending threads makes forward progress in a finite number of time steps of any of the contending threads. A given thread may help other threads complete before it can complete its own operation. Finally, obstruction freedom is the weakest condition of the three and guarantees that a thread will make forward progress only in the absence of contention, in a finite number of attempts on its own. Under obstruction freedom threads makes forward progress only as long as they do not encounter a synchronization conflict from other threads. Additional support may be necessary to either prevent or recover from livelocks.

We define speculative state as all the registers and memory locations a thread modifies inside the transaction block. The speculative state consists of the read set, the write set and any additional information on the status of the actual memory location (state information). The read set refers to the memory locations the thread reads while the
write set refers to locations that may be read or written. In order to successfully execute the transaction, the thread has to reach the end of its atomic block without conflicting with other concurrently executing threads. In other words, there must be no intersection between the write set of one transaction with the read/write set of all other transactions.

If a thread successfully executes the transaction, it can commit its speculative state. The speculative state is then made part of the global state and is now visible to all other threads. The resulting execution is said to be **serializable**, as the result of executing concurrent transactions is similar to a case when these transactions are executed serially.

However, in the presence of conflicts, the speculative state cannot be made visible to other threads. Therefore, the transaction has to be aborted and retried. The threads discard the speculative state and transfer the control to the beginning of the transaction. Since transactions can get aborted indefinitely, the TM implementation must also provide some guarantees for forward progress.

### 2.3.2 TM Design Choices and Implementation Policies

TM implementations can be classified based on the policies for handling interactions between transactional and non-transactional codes [7]. An implementation that supports *weak isolation* allows transactions to execute independent of each other. On the other hand, *strong isolation* guarantees that transactions can execute independent of all accesses, including ones that happen from outside transactions. With strong isolation, individual operations outside the atomic block are treated as though each of those operations execute within an atomic block.

Transactional nesting [42, 47, 48] defines the semantics for nested transactions. In general, the state of outer transactions is visible to inner transactions but the reverse is not always true. With *flat nesting*, nested transactions are treated as one big transaction.
An abort of the inner transaction aborts the outer transaction. With *closed nesting*, inner transactions can abort without terminating their outer transaction. Changes made by inner transactions are visible to outer transactions but are not visible globally until the outer transaction commits. *Open nesting* allows inner transactions to commit globally even if the outer transaction aborts.

Conflict detection of data objects can be performed at word-, cache- or object-level granularity. Supporting fine-grained granularity (word-level or cache-line) is especially useful for aggregate data structures such as arrays, since threads can concurrently access different elements.

From the programmer’s perspective, it is simpler to reason about object-level granularity since the underlying memory layout is hidden. However, languages such as C/C++ have fixed layout for their data structures and modifying these structures to hold the state information is not simple. Further, its harder to support non-transactional legacy codes with object-level granularity than with word- or cache-line granularity [42].

Most implementations of TM use optimistic concurrency control since some of the early TM work was based on non-blocking synchronization. Most hardware implementations leverage caches, coherency protocols and speculative execution capabilities of superscalar processors to support optimistic execution of transactions.

Some implementations (e.g. McRT-STM [54]) allow optimistic reads and pessimistic writes to balance performance and implementation complexity. Since concurrent reads do not cause data races, read operations from multiple threads can proceed concurrently. On the other hand, since concurrent writes cause data races, using exclusive access to perform writes keeps the implementation simple and reduces the amount of wasted work. If the TM system does not keep track of the concurrent reads, then
these *invisible reads* may lead to inconsistent execution. Additional validation techniques are necessary to ensure correct execution.

*Conflict detection* deals with how conflicting operations are detected and is classified into lazy and eager conflict detection. Lazy conflict detection waits for the transaction to complete before checking for conflicts with other transactions. TCC [32] uses a lazy conflict detection scheme. On the other hand, eager schemes detect conflicts early which reduces the amount of speculative work. Most of the hardware schemes such as TLR [52] and UTM [5] support eager conflict detection. Lazy and eager conflict detection techniques are sometimes also called as optimistic and pessimistic conflict detection depending on the context.

*Version management* deals with buffering intermediate results from being visible to other threads until the transaction completes. Version management falls into two categories: Deferred updates (or Lazy version management) and Direct updates (or Eager version management). *Deferred updates* buffers speculative state locally and do not modify the main copy in the memory until the transaction commits. On commit, the object can replaced “in-place” by copying from the private copy or it can be replaced by the private copy entirely.

On the other hand, *direct update* modifies the memory and uses a log to keep track of the original values. This allows for faster commits but slower aborts [19]. The implementation usually uses locks or similar mechanisms to prevent other threads from viewing the uncommitted writes in memory.

*Read versioning* [30, 54] uses global metadata based on version numbers help track the status of memory locations between the time of the first read and each subsequent access (another read, write, or commit). Concurrent write/commit operations from other transactions, with which the reads conflicts, is identified by tracking changes
to these version numbers. On the other hand, *read locks* use locks to implement pessimistic concurrency control for read operations.

2.3.3 Shared Memory Software Transactional Memory

SM-STM designs can be broadly classified into two categories. First, are designs that provide a library–interface to the programmer (Section 2.3.3.1). The second category of designs provide language support and use a compiler to target an underlying STM runtime library (Section 2.3.3.2). We look at each of these in greater detail below.

2.3.3.1 Library Implementations

Shavit and Touitou [56] coined the term *software transactional memory* and proposed lock–free, word-level, static transactions using k-word *compare&swap* operations. Each transaction attempted to acquire ownership of the predetermined memory locations (in increasing order of memory address), using a two-phase locking [29]. Once this operation completes, the transaction is guaranteed to execute to completion without any rollbacks.

Shavit’s STM guarantees lock–freedom by using a non-recursive helping algorithm. Transactions that fail to acquire ownership of a location help transactions that already holds the ownership to complete their operation. Much of the complexity of Shavit’s design comes from the helping algorithm, as it is difficult to determine if any thread is making progress or is blocked.

Each memory word has a corresponding ownership record, called *orec* record. Orec records may point to null (no transactions owns the orec) or may point to the *transaction* record. The transaction primarily holds the transaction’s status, code, and address of memory locations accessed by the transaction.
Herlihy et. al [38] proposed one of the first STM designs to support dynamic transactions, called Dynamic STM (DSTM). DSTM manages a collection of transactional objects that act as wrappers around concurrent data objects. Transactions must first open the object, using the open operation, before it can read/write to it. DSTM uses invisible reads to avoid keeping track of transactions that have read an object. For this reason, a transaction has to validate its read set before it can commit its modifications or before opening another object. The validation step guarantees that no other transaction has attempted to write to the same object.

For writes, DSTM uses an early conflict-detection policy with deferred updates. Obstruction-freedom is used for non-blocking progress and explicit contention managers are used to resolve conflicts. A general interface allows programmers to “plug-in” different contention management policies, with the only constraint that they do not violate the obstruction-freedom guarantee.

Saha et. al [54] propose the McRT-STM that implements transactions using strict two-phase locking and contains commit and abort sequences that are blocking. Since McRT-STM uses locks, it requires a cooperative scheduler to ensure that inactive transactions that may have acquired locks do not block active transactions contending for the same lock. The paper claims simplicity and performance as the main reasons for choosing a two-phase locking scheme. McRT-STM also supports additional features such as nested transactions with partial aborts and conditional signaling within a transaction.

The paper also evaluates various tradeoffs between write-buffering versus undo logging, object versus cache-line conflict detection and read-versioning versus read-write locking. The specific results is highly dependent on the benchmark. Further since McRT-STM requires a cooperative schedule, the effect of pre-emption introduces some unpredictable behavior. The paper presents results for three microbenchmarks based
on simple data structures and an application based on sendmail spam filter. A fine-grained version of the microbenchmarks performs better than McRT-STM using read-versioning and undo-logging. However, McRT-STM performs similarly to fine-grain locking for the sendmail spam filter application.

2.3.3.2 Language Support

Language-based STM implementations have focused on integrating efficient transactional memory constructs into programming languages. Most of the language based efforts have concentrated on:

- Mapping language constructs efficiently on an underlying library-based STM [33].
- Allowing better optimizations that cannot be directly applied to a library-based STM implementation [1, 35].
- Experimenting with transactional semantics and constructs to determine the best solutions that should be exposed to the programmer [33, 34].
- Exploring interactions between transactions and other system functionality such as OS scheduling, exceptions handling, garbage collection, system calls and IO [8, 43].

We look at few of the specific implementations in greater detail. Harris et. al [33] proposed the first language-based STM design called Word-granularity STM (WSTM) and provided support for implicit transactions by extending Hoare’s Conditional Critical Regions (CCR). CCRs provides a guard mechanism that delays the execution of atomic regions until the associated condition evaluates to true. They modified a Java Just-In-Time (JIT) compiler to convert atomic statements to use five primitive STM operations, namely: STMStart, STMAbort, STMCommit, STMValidate and STMWait. References to object fields with atomic regions were also replaced
by appropriate calls to \texttt{STMRead} and \texttt{STMWrite} operations. Supporting word-level
TM eliminated the need to change an object’s layout but complicated the overall design.

Atomos [13] language provides support for implicit open-nested transactions with
strong isolation guarantees. Atomos is based on the Java programming language, re-
placing its synchronization and conditional waiting statements with transactional alter-
natives. In addition to atomic blocks, the language supports: open-nested transac-
tions, software handlers for commit and abort operations, and fine-grained watch sets
and conflict-driven wakesups specified using \texttt{watch-retry} statements. The language
was implemented using the Jikes Research Virtual Machine and the TCC protocol [45].

2.3.4 Distributed Memory Software Transactional Memory

A number of scientific and distributed computing environments require that appli-
cations scale to thousands of nodes in order to benefit from large memory storage and
processing power.

Not surprisingly, researchers have experimented with the idea of extending the
transactional programming model beyond shared-memory multi-core systems [9, 20,
37, 41, 44].

The challenges encountered in designing a scalable STM solutions for GAS and
distributed memory systems is different from those targeting multi-core systems. In
STM designs for shared-memory systems the dominant performance overhead comes
from maintaining metadata required by the STM algorithms themselves. On one hand,
STM algorithms that perform remote communication independent of the application’s
remote data access operations pay a huge price since these additional communication
operations fall on the critical path of a transaction’s completion. On the other hand,
STM algorithms that piggy-back remote metadata operations on top of remote data ac-
cess operations can effectively amortize the overhead of the metadata state maintenance and show good performance/scalability.

Herlihy et al. [37], Manassiev et al. [44], and Kotselidis et al. [41] have all proposed STM designs based on a shared memory view of the globally distributed memory. Adopting such a view allows portability of applications developed for cache-coherent, shared-memory architectures. However, the reliance on global software-based cache coherence fundamentally limits the overall scalability of these implementations to a few dozen nodes.

Recently, Bocchino et al. proposed Cluster–STM [9], the only other STM design specifically designed to target PGAS systems. This is the most relevant work to the ideas presented in this dissertation. Cluster-STM assumes a strict SPMD execution model with no support for thread-level parallelism. At startup, these SPMD processes dedicate a portion of their heap address space for use as a globally accessible Transactional Store (TS). The interface provides transactional semantics only for these TS memory segments of the global address space, thus requiring the client to manage this storage explicitly. Among other things, Cluster–STM makes the following novel contributions to STM designs: (1) Cluster–STM was the first STM design to recognize the need for leveraging RPC mechanisms to exploit locality in distributed STM applications, (2) Similar to distributed database designs, Cluster-STM identified the need to collocate all the metadata corresponding to a memory location within the same node responsible for that location. They recognize this not only as a way to simplify the implementation but also show how this reduces the number of messages required to perform STM book-keeping operations by combining them with data requests, and (3) Cluster-STM evaluated a number of STM algorithms and demonstrated how STM design choices vary between SMSTM and DMSTM schemes.
We believe our GTM runtime library addresses the requirements of PGAS languages better and provides better scalability and performance by: (1) Providing an asynchronous non-blocking protocol to hide the high latency of remote communication operations, (2) Providing a more generalized programming model for large-scale systems that combines SPMD, thread-level parallelism, and RPC mechanisms, and (3) Finally, we guarantees transactional semantics for the entire global address space as long as such accesses use the appropriate STM procedures. This eliminates the need for managing transactional storage explicitly.

In addition to this, none of the current proposals for DMSTM provide language support for specifying transactions. We believe this is very important for the adoption of TM as a viable alternative to lock–based programming.

2.4 Summary

In this chapter we reviewed a number of topics on PGAS models, the Chapel language, and TM relevant to the rest of the dissertation. In particular, we described the limitations of lock–based concurrency control mechanisms in PGAS languages using simple Chapel code examples. We revisit these examples in the rest of this dissertation to make a case for supporting TM based concurrency control mechanisms on large-scale systems.
CHAPTER 3

PARTITIONED GLOBAL ADDRESS SPACE STM TECHNIQUES

We begin this chapter by describing a comprehensive set of design principles for developing scalable STM techniques for PGAS and APGAS languages and runtime libraries in Section 3.1. Next, in Section 3.2 we demonstrate the use of PGAS STM procedures that embody these principles to implement the two sample codes presented in Section 2.2.2. Following this we introduce a novel solution to lowering the cost of remote transactional requests using non-blocking abstractions for STM procedures in Section 3.3. We conclude this chapter by once again demonstrating how such capabilities helps us overlap communication and computation using examples from Section 2.2.2

3.1 Design Requirements and Principles

Today’s distributed STM designs either do not focus on GAS models [20, 37, 41, 44] or do not adequately fulfill the requirements of PGAS languages [9]. STM designs for PGAS models must consider the following aspects in their design/implementation:

1. Support a flexible execution model.
2. Provide transactional semantics across the global address space.
3. Optimize remote communication.
4. Provide an easy to use interface through high-level language constructs.
We now explain these in greater detail. Today’s PGAS/APGAS runtime environments require a portable and general multithreaded programming model. They rely on a static SPMD model wherein a fixed number of SPMD processes are created at startup, disallowing further process creation or migration during the execution of the program. The real flexibility and generality is supported by allowing a dynamic creation and management of user-level threads within each SPMD task. Such an execution model necessitates the need for fully multithreaded STM designs allowing multiple threads of control within a given SPMD process to invoke STM procedures.

In GAS models, the global heap address space is partitioned among SPMD processes as described in Section 2.1. Threads on a given node are allowed direct access to a remote node’s memory space. In order to not impose any specific restrictions on the high-level language or runtime, it is important to provide transactional semantics across this global address space. This not only eliminates the need for managing a separate transactional storage region but also eliminates the need for managing data movement between transactional and non-transactional memory regions.

The dominant performance overhead in SM-STM designs comes from maintaining metadata required by the STM algorithms themselves. On the other hand, distributed STM designs have to factor in the long latency of remote communication operations. In such systems, accessing memory on a remote node is more expensive than the actual cost of manipulating the metadata within that node as part of that data access. Therefore, optimizing remote communication operations plays a vital role in designing a scalable solution.

Fortunately, this is a challenge in any parallel or distributed system. STM designs can thus benefit from the considerable amount of research on optimizing remote communication in these systems. As pointed out by Cluster-STM [9], this includes ag-
gregating remote memory operations, optimizing for data locality, designing STM algorithms that can piggy-back any additional remote metadata operations along remote data operations. In addition to this, one of the most important optimizations for tolerating remote communication latency involves overlapping it with other computation or remote communication work. We later describe how this can be achieved using non-blocking STM abstractions in Section 3.3.

Empirical evidence has shown how using library–based STM designs can be a barrier to the adoption of TM programming, especially by inexperienced programmers using shared-memory STM libraries [53]. In addition to marking the scope of a transaction, STM library designs require that individual load and store operations also be manually identified at the application level using special STM procedures.

To overcome this limitation, researchers working on multicore-STM designs have long proposed language extensions that enable programmers to intuitively mark transactional code segments. Further, they have developed compiler techniques and optimizations that efficiently transforms these code segments to operations supported by an underlying STM runtime library. We believe such language support is also required for the adoption of PGAS STM designs. In this case, the problem is further acerbated since we are dealing with transactional data accesses and RPC calls that perform operations across a distributed-memory system.

Not surprisingly, recent PGAS languages have proposed syntactic extensions for concurrency control in order to simplify the programmer’s task of writing correctly synchronization code. As a matter of fact, all three HPCS languages provide language support for concurrency control, albeit with varying semantics.
3.2 Sample Codes

The purpose of this section is to demonstrate the use of PGAS STM procedures to implement the two sample codes presented in Section 2.2.2. It must be noted that the following discussion is not tied to any specific language or implementation. For example, we can implement these STM procedures using Cluster-STM (Section 2.3.4) or using GTM (Chapter 4).

3.2.1 RA Kernel

We had first presented the Chapel version of RA Kernel in Section 2.2.2.1. The verification loop of the RA kernel uses the atomic construct to guarantee atomic updates to a distributed table $T$:

```plaintext
on T(idx) do atomic T(idx) ^= r;
```

The code listing in Figure 3.1 implements the above atomic code segment using STM procedures. As mentioned earlier, the on construct uses RPC mechanisms to optimize for data-locality. In our example, this results in a purely–local transaction executing on the node that is responsible for $T[idx]$. Thus these STM procedures may look similar to what may usually be supported in a SM-STM library implementation.

The STM procedures $\text{TX-BEGIN (line 5)}$ and $\text{TX-COMMIT (line 9)}$ start and commit a transaction respectively. $\text{TX-BEGIN}$ initializes the transaction descriptor and saves the execution state to revert back to in case of an abort. The $\text{TX-COMMIT}$ attempts to commit the transaction by ensuring that the transaction does not conflict with other concurrently executing transactions and then releases the read/write set. In case of a conflict, one or more transactions are made to restart their execution beginning at $\text{TX-BEGIN}$. 

42
void ra_corep(TDesc* tx, int r, int s) {
    int idxz = r >> (64 - n);
    void* addrTidxz = &myT[idxz - myBase];

    TX_BEGIN(tx);
    temp = TX_LOAD(tx, addrTidx);
    temp = temp + r;
    TX_STORE(tx, &temp, addrTidx);
    TX_COMMIT(tx);
}

Figure 3.1. HPCC RA Kernel: Using fully-local STM Library Procedures

On a PGAS system, each node maintains a segment of array (myT) that is responsible for in a globally addressable region of memory. We calculate the address of the table element corresponding to index idx using relative offsets to the per-node array. We refer to this address as addrTidx, a narrow reference or a regular memory reference.

The STM procedure TX_LOAD (line 3) performs a local transactional load operation from a globally shared address (addrTidx) to a private memory location on the thread’s stack (temp). It keeps track of this operation in the read set of tx. The procedure TX_STORE (line 4) performs a transactional store operation from the thread’s stack memory (temp) to the table entry referred by addrTidx.

3.2.2 RA2 Kernel

The RA2 kernel, first presented in Section 2.2.2.2, uses the atomic construct to guarantee atomic updates of two elements of a distributed table T.

The code listing in Figure 3.2 demonstrates the use of PGAS STM procedures to implement the atomic code segment. Following the terminology used in Chapel,
1 void ra2_core(TDesc* tx, int r, int s) {
2     int idx1 = indexMask(r);
3     int idx2 = indexMask(s);
4     void* addrTidx1 = &myT[idx1 - myBase];
5     int nodeTidx2;
6     void* addrTidx2;
7
8     /* Calculate nodeTidx2 and addrTidx2 */
9     nodeTidx2 = ...;
10     addrTidx2 = ...;
11
12     /* Main Transaction */
13     TX_BEGIN(tx);
14     temp1 = TX_LOAD(tx, addrTidx1);
15     temp1 = temp1 + r;
16     TX_STORE(tx, &temp1, addrTidx1);
17     temp2 = TX_GET(tx, nodeTidx2, addrTidx2);
18     temp2 = temp2 + s;
19     TX_PUT(nodeTidx2, &temp2, addrTidx2);
20     TX_COMMIT(tx);
21 }

Figure 3.2. RA2 Kernel: Using PGAS STM Library Procedures

we refer to the tuple \((\text{nodeTidx}, \text{addrTidx})\) as a \textit{wide reference}. In accordance
with the PGAS memory model, wide-references are memory references that contain a
memory address and the node on which the corresponding data exists. As such, wide-
references may even refer to a locally accessible data object.

The STM procedure \texttt{TX\_GET} (line 17) performs a remote transactional load opera-
tion from the heap memory address \texttt{addrTidx2} on the remote node \texttt{nodeTidx2}
) to local private memory on the thread’s stack. Similarly, the procedure \texttt{TX\_PUT}
(line 19) performs a remote transactional store operation from a temporary variable
in the thread’s stack to a globally shared address \texttt{addrTidx2} on a remote node
\texttt{nodeTidx2}.
These operations default to local transactional operations when the node identified by \texttt{nodeTidx2} also happens to be the node executing the transaction.

Determining the address of the table entry corresponding to \texttt{idx2} may require additional remote communication operations. In order to optimize these additional remote communication operations as well as optimize for data locality, we once again use RPC mechanisms to move “the computation to the location of the data” instead of fetching the data, operating on it locally, and storing it back. This was implemented as follows:

```plaintext
on T(idx1) do atomic {
    T(idx1) ^= r;
    on T(idx2) do T(idx2) ^= s;
}
```

However, unlike the “outer” \texttt{on} construct, the inner \texttt{on} construct has to rely on transactional RPC mechanisms to ensure both the data-accesses are performed in an atomic manner.

The listing in Figure 3.3 uses transactional RPC procedures to send the computation to the remote node \texttt{nodeTidx2} (line 11). The \texttt{updateFn} procedure (line 15) is executed on the remote node local transactional operations. This reduces the number of remote messages from two (Figure 3.2) to one. The \texttt{TX\_COMMIT} (line 12) is additionally responsible for committing transactions that spans multiple locales.

### 3.3 Transactional Asynchronous Execution

This section describes a novel strategy for optimizing the cost of remote transactional requests within a transaction by overlapping them with other requests within the transaction. Section 3.3.1 presents the motivations for supporting \textit{asynchronous} processing of transactional requests. We follow this with a demonstration of how the
```c
1 void ra2_core(TDesc* tx, int r, int s) {
2    int idx1 = r >> (64 - n);
3    int idx2 = s << (64 - n);
4    void* addrTidx1 = &myT[idx1 - myBase];
5    int nodeTidx2;
6
7    // Calculate nodeTidx2
8    TX_BEGIN(tx);
9    temp1 = TX_LOAD(tx, addrTidx1);
10    temp1 = temp1 + r;
11    TX_STORE(tx, &temp1, addrTidx1);
12    TX_FORK(tx, nodeupdateFn, s, idx2)
13    TX_COMMIT(tx);
14 }
15
16 void updateFn(int s, int idx2) {
17    void* addrTidx2 = &myT[idx2 - myBase];
18    temp2 = TX_LOAD(tx, addrTidx2);
19    temp2 = temp2 + s;
20    TX_STORE(tx, &temp2, addrTidx2);
21 }
```

Figure 3.3. RA2 Kernel: Using PGAS RPC STM Library Procedures

sample codes presented in Section 3.2 benefit from such capabilities. We conclude this section with the benefits, limitations, and challenges of leveraging asynchronous protocols in STM designs.

3.3.1 Motivation

Today’s shared-memory as well as distributed STM designs enforce a blocking STM abstraction. When a thread executing a transaction issues a blocking STM request, for example a transactional load operation, it is forced to wait for the underlying runtime implementation to synchronously complete the data/metadata operations necessary to
satisfy the request. In most cases, the thread issuing the request is the one performing the low-level data/metadata operation as well. This not only exposes the cost of transactional accesses the application but also curtails the thread executing the transaction from doing any other useful work. Transactions by definition access and/or update more than one memory location. Using a blocking STM design, threads loses a significant opportunity to proceed with its other operations while blocking on each operation to fully complete. As transactions execute for longer durations, the cost and probability of abort increases.
The problem of waiting on each and every request to complete before proceeding to the next has a more pronounced effect on large-scale systems, since the cost of accessing remote memory is much more expensive than accessing local memory. We believe that to effectively support the TM paradigm on large-scale systems (e.g., Cray XT [3], BlueGene/P [2]), the underlying STM runtime must support techniques that tolerate the long latency of remote data/metadata operations.

A commonly used approach for latency tolerance involves the use of non-blocking procedures to execute the request in an asynchronous manner. Such approaches have long been in use in message passing libraries, one-sided communication libraries, and even distributed databases.

The essential idea is quite simple. A thread issues a request, say a remote get request, and attaches a unique handle to that request. The underlying communication library performs the basic setup to process the request and returns control back to the issuing thread. This effectively allows the thread issuing the request to continue executing other operations and thus overlap the latency of the remote get operation events with useful computation. The handle can be used to block the thread at any point before the thread needs the requested data. Implementations provide various methods to test the status of the handle and wait for the attached request to complete. These designs also impose additional restriction on when handles and input buffers to pending requests can be reused etc.

The same concept can be extended to transactional execution as well. Here, the scope of the transaction imposes certain restrictions on how early such requests can be issued and when they should complete. Figure 3.4 shows how asynchronous execution of remote transactional requests helps hide the latency of communication operations. Unlike in the synchronous case (top half of Figure 3.4), differentiating between when a
transactional request is issued and when it is expected to complete allows a number of computation and communication activities to occur between these two events.

STM procedures need to provide ways to separate the issue and completion of transactional operations. This involves attaching unique handles to transactional operations so as to be able to refer to them later. In addition, this involves providing procedures that test or wait for pending transactional operations to complete.

However, supporting asynchronous transactional requests and non-blocking STM procedures has its challenges. Consider what happens when a transaction issues two requests in a non-blocking manner but one of the requests returns with a failure status. This implies the transaction has to rollback and retry its operation. However, it cannot do so until the other pending operation completes. Therefore, the implementation has to provide mechanisms to block the transaction until all other pending operations have completed.

A second challenge involves the manner in which metadata structures are implemented. Usually, transaction private meta-data is assumed to be accessed by a single thread of control. Having multiple pending requests may potentially attempt to access and modify the metadata in an inconsistent manner. Additionally, we must ensure two conflicting non-blocking requests are not issued from the same transaction.

3.3.2 Sample Code: RA2 Kernel

The code listings in Figure 3.2 and Figure 3.3 demonstrated the use of PGAS-STM procedures to implement the RA2 kernel. In both these cases, the process/thread executing the transaction blocked waiting for the remote node to satisfy the data/metadata request and to send an appropriate acknowledgment back to the source. Since no remote latency is overlapped, the transactions may spend thousands of cycles waiting for
Figure 3.5. RA2 Kernel: Using Non-Blocking PGAS STM Library

Procedures

t these operations to complete. This in turn increases the probability of conflicts and negatively impacts the performance due to transactions being aborted. The problem may be overcome by overlapping these remote communication operations with other requests that do not have any data-dependencies with these requests. The code listing in Figure 3.5 demonstrates the use of non-blocking transactional data access operations.

Here, the procedure TX_GET_NB (line 15) and TX_PUT_NB (line 15) are non-blocking in nature, and transfer data between the global memory space and the temporary storage on the stack asynchronously with respect to the caller. This implies the caller issues remote operations and continues executing the rest of the code instead of waiting for those requests to complete immediately. Thus, TX_GET_NB to nodeTidx2 (line 9) is overlapped with TX_LOAD (line 10) and blocks on TX_WAIT

```c
1 void ra2_core(TDesc* tx, int r, int s) {
2   int idx1 = r >> (64 - n);
3   int idx2 = s << (64 - n);
4   void* addrTidx1 = &myT[idx1 - myBase];
5   int nodeTidx2;
6   handle_t handle1, handle2;
7
8   TX_BEGIN(tx);
9   TX_GET_NB(handle1, tx, nodeTidx2, addrTidx2);
10  temp1 = TX_LOAD(tx, addrTidx1);
11  temp1 = temp1 + r;
12  TX_WAIT(handle1);
13  temp2 = temp2 + s;
14  TX_PUT_NB(handle2, nodeTidx2, &temp2, addrTidx2);
15  TX_STORE(tx, &temp1, addrTidx1);
16  TX_WAIT(handle2);
17  TX_COMMIT(tx);
18 }
```
Figure 3.6. RA2 Kernel: Using PGAS RPC STM Library Procedures

(line 12). The TX_WAIT procedure (line 12 and line 16) ensures the remote request completes before the relevant data is accessed. Similarly the TX_PUT_NB procedure (line 14) is overlapped with TX_STORE on (line 15).

To summarize, code listing in Figure 3.5 differs from the one in Figure 3.2 in the following aspects:

- Declares two handles for keeping tracking of asynchronous requests (line 9/14).
- Uses TX_GET_NB and TX_GET_NB to issue transactional requests in a non-blocking manner.
- Uses TX_WAIT to wait for pending requests to complete.

The same reasoning can be applied to transactional RPC mechanisms as well. The code listing in Figure 3.6 demonstrates the use of non-blocking procedures to implement transactional RPC procedures shown in Figure 3.3. In addition to reducing the number of remote requests and optimizing data-locality, asynchronous transactional RPC mechanisms allow us to overlap the remote request with other STM procedures.

3.3.3 Discussion

As mentioned above, non-blocking STM procedures can free the task/thread making the request to perform other useful operations instead of waiting on remote requests to
complete. However, due to data-dependencies not all requests can be performed in an asynchronous manner. The compiler must be aware of these dependencies when using such procedures.

In this dissertation, we use non-blocking STM procedures to hide the overhead of remote communication. We believe similar mechanisms can be used to lower the scalar overhead of STM operations within a node as well. We plan to explore this area as part of future work.

3.4 Summary

This chapter laid the ground-work for designing scalable PGAS-STM schemes for large-scale systems. We identified a comprehensive set of requirements and design principles for these STM designs. We introduced the novel idea of designing non-blocking STM procedures and show how these can be used to overlap the latency of remote communication requests within a transaction. We used code samples to demonstrate the validity of our claim. In Chapter 4 and Chapter 5 we show how these techniques can be implemented and how they can help provide better concurrency control mechanisms for large-scale systems.
CHAPTER 4

GLOBAL TRANSACTIONAL MEMORY (GTM): A PGAS–STM LIBRARY

This chapter describes *Global Transactional Memory* (GTM), a scalable STM runtime library for PGAS runtime platforms. This chapter begins with a high-level overview of GTM’s functionality in Section 4.1, followed by a detailed look at each of its procedures in Section 4.2. Next, we re-visit the distributed array update example previously discussed in Chapters 2 and 3, and demonstrate how it may be implemented using GTM procedures. Following this we describe some internal details of the GTM implementation in Section 4.4. Finally, we evaluate our prototype for a variety of workloads (Priority Queue, Red–Black Tree, and RA2 Kernel) in Section 4.5.

4.1 Design Overview

*Global Transactional Memory* (GTM) is a scalable STM runtime library for PGAS and APGAS runtime systems. GTM is intended as a proof-of-concept design to showcase the opportunities and challenges involved in implementing scalable STM libraries on large-scale systems. Among other novelties, GTM is the first STM design to support the *non-blocking abstraction* for STM procedures described in Section 3.3.

GTM targets runtime libraries and languages that implement the PGAS memory model, where the shared-heap is distributed across a fixed number of processes that may be running on different system nodes (Section 2.1). In this regard, GTM procedures can
be used to create, execute, and commit transactions that update the shared-heap on one
more more nodes in an atomic manner.

The uniqueness of GTM is that it allows these procedures to execute in a non-
blocking manner (Section 3.3.1). GTM is the first STM design to provide such an
interface and allows transactional requests to execute asynchronously with respect to
the caller.

Transactional semantics are guaranteed across this global address space as long as
such operations occur inside the dynamic context of a transaction and use the appropri-
ate GTM procedures.

Our GTM prototype is implemented using the GASNet communication library [11].
Our implementation is thread-safe and permits multiple threads to issue both syn-
chronous and asynchronous transactional requests using GTM procedures. It is in-
tended to provide a rich set of transactional operations that complements GASNet’s
own Active Messaging (AM), and one-sided Remote Memory Access (RMA) function-
ality.

4.2 GTM Interface Design

Figure 4.1 lists the various GTM procedures. In general, these procedures accept
two special descriptors (or handles): gtm_t.x.t and gtm_op.t. The gtm_t.x.t de-
scriptor represents the transaction itself and is used to store all its private metadata,
including but not limited to its status, read/write set, etc. We use the terms descriptors
or handlers to mean the same thing.

In GTM’s terminology, source node identifies the SPMD process that initiated the
original request and target node identifies the SPMD process on whose context the
operation must be executed.
1 void gtm_init();
2 void gtm_exit();

4 sigjmp_buf *gtm_tx_get_env(gtm_tx_t* tx);
5 gtm_tx_t* gtm_tx_create();
6 void gtm_tx_destroy(gtm_tx_t* tx);

8 gtm_op_t* gtm_op_create();
9 void gtm_op_destroy(gtm_op_t* op);

11 void gtm_tx_begin(gtm_tx_t* tx);
12 void gtm_tx_commit(gtm_tx_t* tx);
13 void gtm_tx_abort(gtm_tx_t* tx);

15 void gtm_tx_malloc(gtm_tx_t* tx, gtm_op_t* op, void* addr,
                     int tgtNode, int size);
16 void gtm_tx_free(gtm_tx_t* tx, gtm_op_t* op, int tgtNode, 
                   void* addr, int size);
17 void gtm_tx_load(gtm_tx_t* tx, gtm_op_t* op, void* dstAddr,
                    int tgtNode, void* srcAddr, int size);
18 void gtm_tx_store(gtm_tx_t* tx, gtm_op_t* op, void* srcAddr,
                    int size, int tgtNode, void* dstAddr);
19 void gtm_tx_fn(gtm_tx_t* tx, gtm_op_t* op, int tgtNode,
                 fnProto_t, void *inbuf, int insize, void *outbuf, int 
                 outsize);

21 int gtm_op_test(gtm_op_t* op);
22 int gtm_op_wait(gtm_op_t* op);
23 int gtm_fn_wait(gtm_op_t* op);

Figure 4.1. List of GTM Procedures
TABLE 4.1

GTM PROCEDURE CALL VARIATIONS

<table>
<thead>
<tr>
<th>Call semantics</th>
<th>op</th>
<th>tgtNode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Blocking</td>
<td>NULL</td>
<td>Source node</td>
</tr>
<tr>
<td>Local Non-Blocking</td>
<td>Valid gtm_op_t</td>
<td>Source node</td>
</tr>
<tr>
<td>Remote Blocking</td>
<td>NULL</td>
<td>Remote node</td>
</tr>
<tr>
<td>Remote Non-Blocking</td>
<td>Valid gtm_op_t</td>
<td>Remote node</td>
</tr>
</tbody>
</table>

The gtm_op_t descriptor acts as a handle for pending asynchronous requests issued in a non-blocking manner. The blocking or non-blocking nature of these GTM procedures is determined by the op argument. If op points to a non-NULL gtm_op_t descriptor then the procedure is non-blocking and is blocking otherwise. Further, the request is considered local if tgtNode is same as the source node issuing the request and is remote otherwise. The combination of op and tgtNode gives rise to four variations, namely local blocking, local non-blocking, remote blocking, and remote non-blocking. We summarize the different ways these procedures can be invoked local vs remote and blocking vs non-blocking in Table 4.1. We describe the various GTM procedures and how they use these descriptors in detail below.

The gtm_init and gtm_exit calls provide the initialization and clean-up functionality. Each SPMD processes must call gtm_init before all other GTM calls and gtm_exit after all other GTM calls.

The procedures gtm_tx_create and gtm_tx_destroy create and destroy a transaction descriptor respectively. The procedures gtm_tx_begin, gtm_tx_commit, and gtm_tx_abort start, commit, and abort transactions on the specified tx. All
these procedures are strictly blocking and local to the node on which they are initiated. However, the `gtm_tx_commit` and `gtm_tx_abort` calls may implicitly initiate remote communication based on whether the transaction performed remote operations or not. The `gtm_tx_abort` additionally ensures all asynchronous requests previously issued inside the transaction complete before the transaction is retired (see Section 4.4).

Nested transactions must use the outermost transaction descriptor and can be created using dynamic pairs of `gtm_tx_begin` and `gtm_tx_commit` calls. Nested transactions are handled by subsuming all inner transactions into the context of the outer-most transaction, namely flat nesting. In other words, an abort at any nested level will abort the entire transaction. A commit at any nested level will be made globally visible only when the outer-most transaction commits successfully.

GTM supports data transfer operations from/to transfer data to/from local/remote transactional storage from/to local private memory with appropriate semantics. The `gtm_tx_load` call copies size bytes from the source address starting at `srcAddr` on `tgtNode` to the destination address starting at `destAddr` within the transaction’s local private address space. The `gtm_tx_store` call copies size bytes from the source address starting at `srcAddr` on the transaction’s private address space to the destination address starting `destAddr` on `tgtNode`’s memory partition.

GTM also supports dynamic management of the global heap storage using the `gtm_tx_malloc` and `gtm_tx_free`. Memory allocated using `gtm_tx_malloc` is automatically freed on an abort and memory freed using `gtm_tx_free` is made visible globally only on a successful commit. In general, these calls are valid only inside the scope of a transaction (i.e. between a dynamic pair of `gtm_tx_begin` and `gtm_tx_commit`). A transaction can commit only if all its data movement and memory allocation procedures execute without conflicting with other concurrently executing
transactions. The non-blocking equivalent of these procedures must be followed by a
\texttt{gtm\_op\_wait} with the corresponding \texttt{op} descriptor (described below).

The procedures \texttt{gtm\_tx\_fn} and \texttt{gtm\_fn} are used to invoke pre-registered user-
level routines. The invoked routines themselves may start their own transactions or in
turn use the \texttt{gtm\_fn} call to invoke other user-level routines or a combination of both.
All \texttt{gtm\_tx\_fn} procedures must be called only inside a transaction, and like nested
transactions use the transaction descriptor of the outer-most transaction. A transaction
can commit only if all its \texttt{gtm\_tx\_fn} procedures execute without conflicting with other
concurrently executing transactions. The non-blocking equivalent of the \texttt{gtm\_tx\_fn}
procedure must be followed by a \texttt{gtm\_fn\_wait} (described below).

The \texttt{gtm\_fn} procedure helps invoke user-level routines that start and commit trans-
actions on the \texttt{tgtNode} node without maintaining any direct dependence with other
transactional/non-transactional operations issued by the same process/thread. It is in-
tended to be used from the non-transactional portions of the code. However, if called
inside the context of another transaction, either directly or indirectly from a routine
invoked via the \texttt{gtm\_tx\_fn} procedure, it executes independently and is not affected
by the eventual fate of the transaction. The non-blocking equivalent of the \texttt{gtm\_fn}
procedure must be followed by a \texttt{gtm\_fn\_wait} with the corresponding asynchronous
handle descriptors.

The procedures \texttt{gtm\_op\_create} and \texttt{gtm\_op\_destroy} create and destroy non-
blocking handle descriptors used for tracking asynchronous requests. During its life-
time, a given handle can be reused for tracking different asynchronous requests initiated
through the non-blocking equivalent of the GTM procedures for data transfer, memory
allocation, and function call invocation. However, at any given instant each \texttt{op} handle
can track only one asynchronous request in-flight.
The `gtm_op_test` procedure queries the execution status of the asynchronous request associated with the `op` handle. The call returns immediately with the one of the following: the request executed without any conflicts, the request conflicted with another transaction, or the request hasn’t completed. This status information can be used to decide if the transaction must abort, attempt to commit, or keep executing the rest of the transaction.

The `gtm_op_wait` procedure must always be used inside a transaction to block for the asynchronous request associated with `op` to complete. If the request executed without conflicts, then `gtm_op_wait` frees `op` for reuse in future non-blocking GTM procedures and makes the data payload (e.g. at `destAddr` in the case of `gtm_tx_load`) available for usage. If the request failed due to conflicts, `gtm_op_wait` implicitly waits for all other pending asynchronous requests to complete before aborting and retrying the transaction.

The `gtm_fn_wait` procedure is used for waiting on non-blocking equivalent of the `gtm_fn` procedure and must always be called outside a transaction. If the invoked routine (`fnName`) started its own transactions and committed without conflicting with any other transactions, then `gtm_fn_wait` frees `op` for reuse in future non-blocking GTM procedures and makes the data payload (i.e. `outBuf`) available for consumption.

4.3 Sample Code: RA2 Kernel

This section presents a variation of the RA2 kernel presented in Section 3.2 and demonstrates how it can be implemented using GTM procedures. Figure 4.2 shows the most straightforward implementation of RA2 kernel. Here we create the transaction on the node issuing the request and issue remote requests to nodes `tgtNode1` and `tgtNode2`. It uses blocking STM procedures and uses GTM’s transactional data ac-


1 gtm_tx_t* tx = gtm_tx_create();
2 gtm_tx_begin(tx);  /* Start Transaction */
3 gtm_tx_load(tx, NULL, tgtNode1, &temp1, &T[idx1], sizeof(temp1));
4 temp1 = temp1 + val1;
5 gtm_tx_store(tx, NULL, tgtNode1, &temp1, sizeof(temp1), &T[idx1]);
6 gtm_tx_load(tx, NULL, tgtNode2, &temp2, &T[idx2], sizeof(temp2));
7 temp2 = temp2 + val2;
8 gtm_tx_store(tx, NULL, tgtNode2, &temp2, sizeof(temp2), &T[idx2]);
9 gtm_tx_commit(tx);  /* Commit Transaction */
10 gtm_tx_destroy(tx);

Figure 4.2. RA2 Kernel: Using GTM’s Transactional Load and Store Procedures

cess procedures to first fetch data from T(idx1), update it locally, and store the value back in T(idx1). The same process is repeated for T(idx2). val1 and val2 represent two random values.

As explained in Section 3.2, this kernel can benefit from non-blocking STM procedures to help tolerate the latency of communication operations. Figure 4.3 demonstrates how non-blocking GTM procedures can be used to implement the RA2 kernel. As observed earlier, given the lack of data dependency the two fetch operations and store operations can proceed in parallel.

Figure 4.4 implements the RA2 kernel using non-blocking gtm_tx_fn procedure in order to optimize for data locality. This is the most optimized implementation of RA2 kernel among all the cases presented. Using the RPC mechanism helps eliminate two remote messages, one for each data access. Moreover, the non-blocking implementation helps overlap the two communication operations with each other.
1 gtm_tx_t* tx = gtm_tx_create();
2 gtm_op_t* op1 = gtm_op_create();
3 gtm_op_t* op2 = gtm_op_create();
4 gtm_tx_begin(tx); /* Start Transaction */
5 gtm_tx_load(tx, op1, tgtNode1, &temp1, &T[idx1], sizeof(temp1));
6 gtm_tx_load(tx, op2, tgtNode2, &temp2, &T[idx2], sizeof(temp2));
7 gtm_tx_wait(tx, op1);
8 temp1 = temp1 + val1;
9 gtm_tx_store(tx, op1, tgtNode1, &temp1, sizeof(temp1), &T[idx1]);
10 gtm_tx_wait(tx, op2);
11 temp2 = temp2 + val2;
12 gtm_tx_store(tx, op2, tgtNode2, &temp2, sizeof(temp2), &T[idx2]);
13 gtm_op_wait(tx, op1);
14 gtm_op_wait(tx, op2);
15 gtm_tx_commit(tx); /* Commit Transaction */
16 gtm_op_destroy(op1);
17 gtm_op_destroy(op2);
18 gtm_tx_destroy(tx);

Figure 4.3. RA2 Kernel: Using GTM’s Non-Blocking Transactional Load/Store Procedures.
1 gtm_tx_t* tx = gtm_tx_create();
2 gtm_op_t* op1 = gtm_op_create();
3 gtm_op_t* op2 = gtm_op_create();
4 inBuf1.index = idx1;
5 inBuf1.amount = val1;
6 inBuf2.index = idx2
7 inBuf2.amount = val2;
8 gtm_tx_begin(tx); /* Start Transaction */
9 gtm_tx_fn(tx, op1, tgtNode1, FN_TRANSFER, inBuf1, sizeof(inBuf1), NULL, 0);
10 gtm_tx_fn(tx, op2, tgtNode2, FN_TRANSFER, inBuf2, sizeof(inBuf2), NULL, 0);
11 gtm_fn_wait(tx, op1);
12 gtm_fn_wait(tx, op2);
13 gtm_tx_commit(tx); /* Commit Transaction */
14 gtm_tx_destroy(tx);

16 void transfer(gtm_tx_t* tx, int index, int amount) {
17      gtm_tx_load(tx, NULL, here, &temp, &T[index],
18                  sizeof(temp));
19      temp = temp + amount;
20      gtm_tx_store(tx, NULL, here, &temp, sizeof(temp),
21                     &T[index]);
20 }

Figure 4.4. RA2 Kernel: Using GTM’s Non-Blocking Transactional RPC Procedures
4.4 GTM Implementation

Our GTM prototype is implemented using the GASNet communication library [11]. Our implementation is thread-safe and permits multiple threads to issue both synchronous and asynchronous transactional requests using GTM procedures. It is intended to provide a rich set of transactional operations that complements GASNet’s own Active Messaging (AM), and one-sided Remote Memory Access (RMA) functionality. Furthermore, the prototype has been implemented to coexist with GASNet on the same software stack layer, allowing current PGAS languages already using GASNet to seamlessly use GTM for implementing language-level TM constructs.

The core STM algorithm is based on a word-based, read versioning [30, 54], deferred update [33, 38], and eager acquire [27, 35] scheme. We use Hans Boehm’s atomic_ops library [10] to implement memory barriers, single-word atomic swap, and single-word atomic increment/decrement operations. We specifically target 64 bit architectures and currently require all multi-word operations to be a multiple of 8 bytes. Both these design choices did not affect the workloads we studied.

As mentioned earlier, we assume the weak-isolation model. Therefore, we track conflicts only among transactional accesses to memory. We do not track the conflicts between transactional and non-transactional accesses. We assume program phases are separated by global barriers and allow memory to be accessed using different modes across these phases. However, within a given phase of the program memory locations may be accessed from inside a transaction through GTM procedures or from outside transactional boundaries through some other mechanism, including GASNet, but not both. Many aspects of our implementation are inspired by and based on the design principles of a number existing STM designs [9, 24, 30] and distributed database designs [6, 28]. We describe the different components of our implementation below.
4.4.1 Metadata Organization

GTM maintains a variety of metadata information for guaranteeing the necessary transactional semantics. This includes: per-node shared metadata, per-transaction private metadata, and per-node shared transaction descriptor table.

4.4.1.1 Per-Node Shared Metadata

Word-based STMs maintain a hash-table, typically referred to as ownership records (ORecs), for storing state information accessed by all transactions. This metadata is checked as part of every transactional operation to determine the status of the memory word it is trying to access. Each hash-table entry stores metadata for some unique set of memory locations in the node’s heap address space. The least significant bit (LSB) of each entry is set if any of the memory addresses that maps onto it are currently being written. We refer to this bit as the write bit. Otherwise, the entry maintains a 63-bit version number for the memory addresses.

In our case, each node maintains a globally-accessible hash-table (2^{20} entries per node) for this purpose. Remote transactional operations, similar to local ones, first check this hash table before proceeding with their data access. We describe the exact algorithm in detail below.

The structure of this hash-table potentially leads to false sharing. In other words, transactions that are accessing different memory locations may conflict with each other because these memory locations map to the same hash-table entry. This is a general problem faced in all word-based STMs. We are currently investigating alternatives to overcome this problem.
4.4.1.2 Per-Transaction Descriptor Metadata

As mentioned in Section 4.2 the transaction descriptor data-structure is private to a given transaction and is used to store the state information that is associated with that transaction. We maintain the following metadata in each transaction’s descriptor: transaction’s status, read/write set, list of remote nodes, list of pending asynchronous requests, an implicit gtm_op_t handle.

The transaction’s status indicates whether the transaction is idle, actively executing, in the commit phase, or being aborted. The transaction’s nesting level indicates the depth of nesting for the current transaction. We currently support flat nested transactions and increment this counter as every nested transactional begins operation. We decrement this counter for every transactional commit, but attempt to commit the transaction only for the outermost transaction (nesting_level = 0).

The read/write set keeps track of the list of locations read/written by the transaction. The read set/write sets are implemented as two separate list-based data-structures. Each entry of the read set contains the following metadata: address of the ORec entry corresponding to the memory address being loaded and version number stored in the ORec entry at the time of the load. Each entry of the write set contains the following: destination memory address of the store operation, address of the ORec entry corresponding to the destination memory address, value to be stored, version number stored in the ORec entry at the time of the store, and pointer to the next entry in the write set that maps to the same lock.

The remote nodes list maintains a list of remote nodes targeted by transactional requests initiated by this transaction. The pending requests list maintains a list of non-blocking request handles used by this transaction. This is used to track pending and completed non-blocking requests initiated by the transaction. In case of an abort, trans-
actions use this list to wait on pending non-blocking requests to finish. Only then can
the transaction retry its operation.

The implicit handle is used to implement synchronous blocking GTM procedures. For GTM procedures that have a non-blocking as well as blocking variant, we use the former to implement the latter. For example, the blocking gtm_tx_fn procedure is essentially a wrapper around the non-blocking gtm_tx_fn_nb procedure. We achieve the blocking semantics by immediately calling gtm_fn_wait before we return from the outer gtm_tx_fn procedure call. This helps us help streamline how we wait on other pending requests to arrive in case we need to abort the transaction. Since non-blocking requests require a gtm_op_t descriptor and each transaction have issue at most one blocking request at a given time, we use the implicit gtm_op_t handle for this purpose.

The free on abort and free on commit are two lists that keep track of dynamic memory allocations and frees called from within a transaction. The first list keeps track of all memory blocks that need to be freed on an abort, while the second keeps track of all memory blocks that need to be freed on a successful commit.

4.4.1.3 Interactions Between Different Metadata

To summarize, the per-node shared metadata is responsible for keeping track of state information that all transactions need to be aware of in order to detect conflicting operations from other transactions. The per-transaction private descriptors help transactions maintain a “snapshot” of the required metadata for future use. In GTM, transactions can perform both local and remote operations. This implies the transaction has to keep track of the metadata of remote memory locations as well. This can either be done at the original source node or at the node that is responsible for the data. The
former approach requires us to transfer metadata as part of the data access or as part of
the commit/abort operation. Rather we take the latter approach, originally advocated
by distributed database implementations and also followed in Cluster-STM, where we
maintain the metadata pertaining to a given nodes on that node itself. In other words,
when transactions access data on other nodes, they maintain a transaction descriptors
on each of those nodes to hold their private metadata.

To make this possible, each node maintains a *shared transaction descriptor table*
to enable transactions to create and maintain its state in a distributed manner across
different nodes. This allows us to *slice* the transaction descriptor across all the node on
which it has performed operations on. Each row of this table represents a unique node.
Each entry in that row represents a unique transaction.

When a transaction $t$ is created on node $i$, it attempts to find a free entry in the
$i_{th}$ row of this table. Say this is the $j_{th}$ entry. The tuple $(i, j)$ represents a globally
unique identifier for this transaction. This operation requires a lock since multiple
threads can attempt to do this at the same time. The first time the transaction $t$ sends
a request to a remote node $k$, it allocates a new descriptor and stores in the $i$th row
$j$th column of the descriptor table on the remote node. Transaction $t$ will be the only
transaction accessing that entry and so there is no need for any synchronization in this
case. For each subsequent request, the transaction finds the descriptor at this location.
Finally when the transaction is destroyed it reclaims this storage. As an optimization,
transactions that are marked purely local don’t need to be inserted into this table in the
first place.

The primary advantage of using this data-structure is that we do not have to acquire
locks on the remote node. In general, this is one aspect of our implementation that
requires improvement. For starters, this design does not use memory efficiently and
makes the assumption that transactions always perform remote operations on all the nodes. The other issue here is the initial size of each row. The size of each row must be at least the number of threads per node. This is because each thread can create at most one transaction at a time. However, this is inadequate if we allow non-blocking \texttt{gtm\_fn} and \texttt{gtm\_tx\_fn} procedures to be arbitrarily nested. However, we don’t allow such operations for other limitations (described below).

4.4.2 STM Algorithm

The core STM algorithm consists of two parts: \textit{local–sync} and \textit{remote–async}. The local–sync component implements the GTM procedures that are both local and synchronous in nature. The remote–async component implements GTM procedures that are both remote and non-blocking. We rely on GASNet’s Active Messaging (AM) capability to implement this component. We describe these in greater detail below.

4.4.2.1 Local–Sync Component

The \textit{local–sync} component implements the GTM procedures that are both local and synchronous in nature.

Our current prototype does not support non-blocking local GTM procedures. This would require allowing multiple threads to update the transaction descriptor in a threadsafe manner. We plan to explore this aspect of our design as part of future work. Transaction begin is implemented as follows:

1. Increment the transaction’s nesting depth (\texttt{nestLevel}).

2. If \texttt{nestLevel} == 1, we are beginning a new transaction. Set the status of the transaction to \textit{active} and return.

3. If \texttt{nestLevel} > 1, we are executing a nested transaction. No additional action is required.
The blocking local transactional load operation proceeds as follows for each 8 byte memory word:

1. Atomically read the ORec entry corresponding to the address of the source memory location.

2. If the write bit is set and a write set entry is found in the write set corresponding to the address of the load then return the value from the write set entry.

3. If the write bit is set and a write set entry is found but not for the address to be loaded then return the value from memory.

4. If the write bit is set and no entries are found on the write set for this ORec then it implies some other transaction is currently owning this ORec. Abort the transaction because this signals a conflict.

5. If the write bit is not set then create a new read set entry. Read the value from memory and store it in the read set entry. Store the version number from the ORec in the read set entry. Return the value read from memory.

The blocking local transactional store operation proceeds as follows for each 8 byte memory word:

1. Atomically read the ORec entry corresponding to the address of the destination memory location.

2. If the write bit is set and a write set entry is found in the write set corresponding to the address of the store then store the new value in write set entry.

3. If the write bit is set and a write set entry is found but not for the address to be loaded then create a new write set entry and add it to the write set. Add the following to the new write set entry: address of the ORec, version number as stored in the previous write set entry, value to be stored, and destination address of the store operation. Finally, link the new write set entry to last entry in the list of write set entries in the write set that share the same ORec.

4. If the write bit is set and no entries are found on the write set for this ORec then it implies some other transaction is currently owning this ORec. Abort the transaction.

5. If the write bit is not set and we have a read-set entry corresponding to the destination address in the read set then validate the read set entry. This involves checking to see if the version number of the read set entry is older than the one
currently read in Step 1. If so abort the transaction. If not, then create a new
write set entry as mentioned in Step 3 and add it to the write set. This will be the
first entry in the write set for this ORec and will not require to be linked to other
entries in the write-set.

6. If the write bit is not set and we do not have a read-set entry corresponding to
the destination address then simply create a new write set entry and add it to the
write set.

The blocking local transactional malloc operation proceeds as follows: allocate the
required memory and add it to the free on abort list. The blocking local transactional
free operation proceeds as follows: add the memory to be freed to the free on commit
list.

Transactional commit operation happens in two phases. In the first phase, we ac-
quire a time-stamp that is unique within the current node and validates the read-set.

The validation of the read set proceeds as follows. For each entry in the read-set,
read the ORec entry from memory using the address stored in that read-set entry. If the
ORec entry has the write bit set then abort the transaction if this transaction doesn’t own
that ORec (absent in its write set). If there is no write bit set and the version number do
not match then abort the transaction. This basically implies that some other transaction
is currently writing to or has already updated one of the memory locations that maps to
the same entry in the metadata array. If not the read set entry is valid.

The second phase of the commit operation stores the values of the write-set entries
in the corresponding memory locations and releases the ORec by writing the time-stamp
obtained in the first commit phase into the ORec. This operation always succeeds.

The blocking local transactional abort operation aborts the transaction by setting the
transaction’s status to aborted, discarding the read-set, releasing the entries in the write-
set by unseting the write bit in the ORec entries, resetting the rest of the transaction’s
state to its start state and returning to the start of the transaction using longjmp.
4.4.2.2 Remote–Async Component

The remote–async component implements GTM procedures that both remote and non-blocking. We rely on GASNet’s Active Messaging (AM) capability to implement this component.

For example, \texttt{gtm\_tx\_get\_nb} is implemented as follows:

1. Add target node to the transaction’s remote node list.
2. Send a GASNet AM to remote node to invoke get handler. Continue executing the rest of the transaction until it hits a \texttt{gtm\_op\_wait} for this handle. Jump to Step 5.
3. The Get handler first locates the transaction descriptor corresponding to the source transaction that issued this request. If this is the first request, it creates a new descriptor and stores it in the shared descriptor array.
4. The handler calls \texttt{gtm\_tx\_load}. If the operation succeeds it returns data to the source node. Else it sends a message to the source node to abort the transaction.
5. When the source transaction reaches the \texttt{gtm\_op\_wait} it checks the message. If the data is returned, the transaction continues with its operation and uses it data.
6. If the operation fails, then it waits for all pending asynchronous operations to complete and then aborts the transaction.

The procedures \texttt{gtm\_tx\_put\_nb}, \texttt{gtm\_tx\_malloc\_nb}, and \texttt{gtm\_tx\_free\_nb}, are similar to the \texttt{gtm\_tx\_get\_nb} but call the corresponding put, malloc, and free handlers. These handlers in turn calls the local–sync version of these calls.

The GTM procedure \texttt{gtm\_tx\_fn\_nb} proceeds as follows:

1. Add target node to the transaction’s remote node list.
2. Send AM to remote node to invoke fork handler, and continue executing the rest of the transaction until it hits a \texttt{gtm\_op\_wait} for this handle. Jump to Step 5.
3. The fork handler first locates the transaction descriptor corresponding to the parent transaction that issued this request. If this is the first request, it creates a new descriptor and stores it in the shared descriptor array. It then does a \texttt{setjmp} to store the state in order to jump back on an abort.
4. The handler then invokes the specified user-level routine on the remote node. When any of the local operation fails, it uses longjmp to the point of set jmp and then return to the source transaction with a failure status.

5. When the source transaction reaches the gtm_op_wait it checks the message. If the data is returned the transaction continues with its operation and uses it data.

6. If the operation fails, then wait for all pending asynchronous operations to complete and abort the transaction.

The GTM procedure gtm_fn is a simple wrapper around the AM mechanism.

The commit operation is similar to the get operation. It invokes the first phase of commit on all the nodes in the transaction’s remote node list. We abort the transaction if any of these operations failed. Next we invoke the second phase of commit on all the nodes. The second commit phase always succeeds.

The abort operation waits for all remote requests to complete and then similar to the get operation it invokes the local–sync abort on all nodes in the transaction’s remote node list. Once this is done, it re-executes the transaction from the beginning.

As described earlier, remote blocking procedures are implemented using the remote non-blocking version. In other words, gtm_tx_get calls gtm_tx_get_nb with the implicit handle (Section 4.4.1)

The prototype does not allow gtm_tx_fn and gtm_fn procedures to invoke routines that initiate additional remote communication. This is not a limitation for the workloads presented in this chapter. One possible solution to this problem is to have a pool of threads on each node to execute tasks and have the gtm_tx_fn and gtm_fn procedures insert them into the task queue. We also disallow multiple in-flight non-blocking requests to the same target node.
### TABLE 4.2

**ORNL JAGUAR XT4 SYSTEM CONFIGURATION PARAMETERS**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of compute nodes</td>
<td>7832 (excluding dedicated login nodes)</td>
</tr>
<tr>
<td>Processor Type</td>
<td>Quad-core AMD Opteron Budapest Processors</td>
</tr>
<tr>
<td>Processor Frequency</td>
<td>2.1 GHz</td>
</tr>
<tr>
<td>Processors per node</td>
<td>1</td>
</tr>
<tr>
<td>Total number of cores</td>
<td>31328 cores</td>
</tr>
<tr>
<td>Memory per node</td>
<td>16 GB of DDR2 memory</td>
</tr>
<tr>
<td>Router</td>
<td>Cray Seastar2</td>
</tr>
<tr>
<td>Interconnect Topology</td>
<td>3D Torus</td>
</tr>
<tr>
<td>Peak Router Bandwidth</td>
<td>45.6 GB/s</td>
</tr>
<tr>
<td>Peak System Performance</td>
<td>263 teraflop/s</td>
</tr>
<tr>
<td>Operating System</td>
<td>Compute Node Linux (CNL) micro-kernel</td>
</tr>
</tbody>
</table>

4.5 Evaluation

In this section we evaluate the scalability and performance aspects of our GTM prototype using three workloads: Red–Black Tree, Priority Queue, and RA2 Kernel. These workloads are based on benchmarks used in prior STM work targeting multicore/SMP systems [24, 30].

We performed all our experiments on the NCCS Jaguar Cray XT4 system at ORNL. The Jaguar XT4 system was the fastest systems on the Top 500 list [57] at the time.
these experiments was performed. Each node of the XT4 system consists of a 2.1 GHz Quad-core AMD Opteron processor with 8GB of memory and a Cray Seastar router. Nodes are interconnected in a 3D torus topology, providing very low latency messaging between nodes. We list some of the important hardware and software system configuration parameters of this system in Table 4.2.

All workloads were compiled using PGI-7.2.4 C language compiler (with -O3 optimization) and GASNet (version 1.12.0) Portals Conduit. Each Jaguar XT4 quad-core node is assigned a single SPMD process and each SPMD process is single-threaded.

For each workload, we present two different sets of graphs. In the Execution Time graphs, the X-Axis represents the node count and the Y-Axis represents the total execution time for each problem size (S, L, and M). The problem size identifies the total successful transaction commits across the entire system. The problem is equally distributed amongst all the nodes and each node commits (problem size / node count) transactions. As we increase the node count for a given problem size, each node commits fewer transactions keeping the sum total constant. The random number generator follows uniform distribution and is controlled to provide comparable results across two different runs of a given data point. The use of blocking versus non-blocking GTM procedures is termed GTM_B and GTM_NB respectively for convenience. On a single node, GASNet does not differentiate asynchronous requests from synchronous requests resulting in similar execution times for GTM_B and GTM_NB.

In the Asynchronous Speedup graphs, the X-Axis represents the node count the code was run on and Y-Axis represents the speedup (larger the better) and is calculated by the ratio of execution time for the GTM_B case over GTM_NB case for a given problem size and node count. For example, Speedup-S = <GTM_B,S> / <GTM_NB,S>.
4.5.1 Red–Black Tree

Figure 4.5. Performance of Red–Black Tree Kernel: Execution time and Asynchronous Speedup results.
This workload operates on a distributed RB-tree, based on STAMP’s version of the TL2 RB-tree code [12]. Each process randomly picks a target node and uses the \texttt{gtm\_fn} procedure to invoke RB-Tree routines (insert, delete, and search) on the chosen target. These routines start their own transactions and atomically access the local RB-Tree on the target node using local/blocking transactional loads/stores. Each \texttt{gtm\_fn} results in the successful execution of a single transaction. Transactions are short running with small read/write sets. Here, $S$ is $2^{24}$, $M$ is $2^{28}$, and $L$ is $2^{30}$ committed transactions respectively.

Figure 4.5 presents the execution time and speedup results for the Red–Black Tree workload. \texttt{GTM\_B} and \texttt{GTM\_NB} represent the blocking and non-blocking use of the \texttt{gtm\_fn} procedure. The \texttt{GTM\_NB} case allocates an array of 32 asynchronous handle descriptors at startup, allowing at most 32 simultaneous in-flight asynchronous \texttt{gtm\_fn} requests for each SPMD process.

A number of general observations can be made on the scalability of the Red–Black Tree kernel based on the execution time results. First, the execution time almost halves as the node count is doubled, implying good strong-scaling behavior. For example, the total execution time for $<GTM\_NB,M>$ on 256 nodes (27.73 seconds) is almost half as of 128 nodes (54.33 seconds). Second, the total execution time remains almost the same if the node count and problems sizes are multiplied by the same factor, implying good weak-scaling behavior. For example, the execution time for $<GTM\_B,S>$ on 32 nodes (29.53 seconds) is close to $<GTM\_B,M>$ on 512 nodes (28.71 seconds), given problem sizes $S$ and $M$ vary by a factor of 16. It should be noted that the strong and weak scaling trends do not hold perfectly across all the problem and system sizes tested.

Figure 4.5 also illustrates the real benefits of latency toleration techniques in STMs for distributed memory systems. These techniques provide additional speedups of $2X$
or more for problem size $L$ (Speedup-$L$ curve), speedups of $1.83X$ or more for problem size $M$ (Speedup-$M$ curve), and finally speedups greater than $1.79X$ for problem size $S$ (Speedup-$S$ curve beyond four nodes).

4.5.2 Priority Queue

This workload operates on a distributed priority queue implemented using a sorted linked-list. The test harness is similar to the Red–Black Tree workload. Each process randomly picks a target node and uses the gtm_fn procedure to invoke priority queue routines ($\text{insert()}$, $\text{delete()}$, or $\text{search()}$) on the chosen target. These routines start their own transactions and atomically access the local priority queue on the target node using local/blocking transactional loads/stores. Thus, each gtm_fn results in the successful execution of a single transaction. Transactions are long running with large read/write sets. Here, $S$ is $2^{24}$, $M$ is $2^{28}$, and $L$ is $2^{29}$ committed transactions respectively. Figure 4.6 presents the execution time and speedup results for the Priority Queue workload. Here, $\text{GTM}_B$ and $\text{GTM}_NB$ represent the blocking and non-blocking use of the gtm_fn procedure.

A number of general observations can be made on the scalability of the Priority Queue workload based on the execution time results in Figure 4.6. First, good strong-scaling and weak-scaling attributes can be observed for most problem sizes and node counts. This implies GTM scales well on large node counts. Second, the execution time for $<\text{GTM}_B,S>$ increases going from one to two nodes and is a result of over-the-network accesses that begins to occur at two nodes. Unlike in $<\text{GTM}_NB,S>$, the overhead of remote communication is fully exposed due to the lack of latency tolerance techniques. But as we increase the node count, the work per node decreases, resulting in better performance compared to the one/two node case. Finally, $<\text{GTM}_B,M>$ and
Figure 4.6. Performance of Priority Queue Kernel: Execution time and Asynchronous Speedup results.

<GTN_NB,L> have similar execution times (on 128 nodes it is 98.8s and 102.4s respectively). This implies leveraging asynchronous protocols provide sufficient speedup
to run larger problems on smaller systems if the problem fits in the memory of fewer nodes.

Figure 4.6 demonstrates the significant improvements in performance that can be achieved by issuing transactional operations asynchronously. These techniques provide additional speedups of $1.89X$ or more for problem size $L$ (Speedup-$L$ curve), speedups of $1.75X$ or more for problem size $M$ (Speedup-$M$ curve), and finally speedups greater than $1.76X$ for problem size $S$ (Speedup-$S$ curve beyond four nodes).

4.5.3 RA2 Kernel

This workload implements the version of RA2 kernel discussed in Section 4.3. Each process starts a transaction, randomly picks two target nodes (one of which may be local), and uses $\text{gtm\_tx\_fn}$ to invoke the $\text{transfer\_fn()}$ routine (Figure 4.4) on the chosen target nodes. Unlike the earlier workloads, there can be at most two outstanding operations inside a transaction and one transaction per process. In effect, these transactions are short running with small read/write sets. Here, $S$ is $2^{24}$, $M$ is $2^{28}$, and $L$ is $2^{29}$ committed transactions respectively.

Figure 4.7 presents the execution time and speedup results for the RA2 Kernel. Again good strong-scaling and weak-scaling attributes can be observed from the execution time results. The speedup results exhibit completely different trends compared the other workloads, with speedups ranging between $1.27X$ and $1.38X$. Here, a process executes only one transaction at a time with at most two outstanding operations inside a transaction.
Figure 4.7. Performance of RA2 Kernel: Execution time and Asynchronous Speedup results.
4.5.4 Additional Experiments

We also investigated the effect of the number of threads per SPMD process on performance. On smaller node counts, having two threads per SPMD process provided better execution times than the single thread case. As we scaled to larger node counts, the overhead of communication, the limitations in the number of concurrent in-flight requests limited the benefits of the second computational thread.

4.6 Summary

This chapter presented GTM, a scalable library-based STM design for large-scale systems. GTM is the first of its kind STM library to provide the non-blocking STM abstraction. We described GTM’s interface design and presented details on our implementation prototype. We evaluated GTM for a number of workloads and demonstrated the scalability and performance advantages of using asynchronous transactional requests.
CHAPTER 5

GLOBAL ATOMIC TRANSACTIONS IN CHAPEL

Chapel is the first language that has proposed to expose the concept of distributed atomic transactions to the programmer and thus provides an excellent vehicle for exploring the productivity benefits of TM techniques for large-scale parallel applications. This chapter presents the requirements, methodology, and challenges in implementing global atomic transactions in the Chapel language.

We begin this chapter by describing the syntax and semantic issues surrounding Chapel’s `atomic` keyword in Section 5.1. Our implementation of these semantics required a number of changes to Chapel’s compiler and runtime system (Section 2.2.3).

Our implementation consists of four main components. First, we added a new pass to the Chapel-to-C compiler (Section 5.2) to handle code segments that are enclosed within `atomic` statements. The second component, described in Section 5.3, is the implementation–independent STM interface targeted by the Chapel-to-C compiler. Next, we describe our current implementation of this interface, called `Global Transactional Memory 2` (GTM2), in Section 5.4. Finally, we describe the changes required all the support functionality required to make the above changes work correctly in Section 5.5. We describe each of these in greater detail below.
5.1 Language Semantics

In Chapel, similar to other PGAS languages, the program heap is distributed across multiple independent memories. As explained earlier, this sharing of the address space between threads necessitates the need for concurrency control mechanisms that synchronize concurrent accesses to shared data.

As such, Chapel’s *global-view abstraction* allows the programmer to describe his/her algorithm without having to reason about the physical location of data. This necessitates the need for concurrency control mechanisms that enable the programmer to atomically update data that may be physically distributed across the system. Chapel’s *atomic* statement provides an intuitive interface for specifying such operations.

The basic syntax of the atomic keyword in Chapel is as follows:

```chapel
atomic statement;
atomic { statement-block };
```

The most unique aspect of Chapel’s *atomic* construct is that it allows the atomic update of shared data that may be physically distributed across multiple locales. Chapel’s *atomic* statement extends the global-view abstraction to atomic updates to memory. It provides the programmer a intuitive interface to specify *global transactions* that may execute on a single locale or multiple locales depending on how data is distributed. *Single–locale* transactions execute on the same locale as the task executing the transaction and only access the shared heap on that locale. On the other hand, *multi–locale* transactions or *distributed* transactions, access data on more than one locale and thus require the capabilities of a PGAS STM design for its implementation.

The *atomic*–statement is orthogonal to the rest of the Chapel language, which is still under development. We believe it is too early to strictly define the semantics of a language keyword without understanding its interactions with the rest of the language.
Thus, our current definition of the semantics for $atomic$–statements is mainly driven by the need to balance programmability and performance concerns.

We used a wide-range of Chapel programs to help understand the programmability requirements. Additionally, our earlier work on GTM (Chapter 4) provided us with a better understanding of design decisions that improve or deter performance. We have also taken into account the related work that implement atomic transactions in languages such as C, C++, and Java.

We divide our discussion into two parts. First, we present the basic design features and semantics of $atomic$ statements. Next, we present some of the more advanced features that may be required for a more complete implementation. As such these features are not fully implemented due to the lack of compelling use cases and/or implementation challenges that require additional investigation.

5.1.1 Basic Design

Overall, our current specification of the semantics of $atomic$ statements provide global transactions with weak atomicity and flat-nesting. We permit memory operations and function calls within $atomic$–statements. The atomic construct can be combined with $on$ statements for specifying locality of remote as well as distributed transactions. We disallow I/O operations and blocking synchronization operations within $atomic$–statements. We now describe each of these in greater detail.

5.1.1.1 Control Flow

The $atomic$ keyword can be added before a single statement or a statement block. In the latter case, the transaction begins its execution at the first statement of the statement block, and continues serially until it reaches the end of the statement block. The
effect of the atomic construct extends beyond the static scope of the block statement. In particular, it extends to function calls and \texttt{on} statements in its dynamic scope. From the compiler’s viewpoint, it must generate the correct STM library call for all memory operations that operate on the shared heap. We will see how \texttt{on} statements can be combined with atomic statements in greater detail in Section 5.1.1.4. In the following code snippet, the dynamic scope of the atomic keyword (in line 3) extends to all the operations within \texttt{fun2} as well:

```python
def fun1() { atomic fun2(); }

def fun2() { do_something; }
```

One of two things may occur during the execution of the transaction. The transaction may reach the end of the block statement. This would mean the transaction can attempt to commit its speculative state. The actual operation during the commit phase is implementation dependent. The language semantics as such is not tied to a specific implementation. On the other hand, transactions may abort due to conflicts. In the case of an abort, the transaction rollbacks it state and starts from the first statement in the atomic statement block. Again, how and when conflicts are detected are not enforced by the language semantics.

Chapel does not have C/C++ style \texttt{goto} statements. This implies control cannot enter into the atomic statement block at an arbitrary point. The only entry point into the atomic block is via top of the atomic block. Control flow may be altered inside functions and looping constructs using \texttt{return}, \texttt{break}, and \texttt{continue} statements. These statements can be used within \texttt{atomic}–statements and are required to commit the transaction before returning control to the intended target location.
5.1.1.2 Atomicity Guarantees

Our specification guarantees global atomic transactions will execute in an atomic manner with respect to each other. This implies we support weak-atomicity semantics. This also implies atomic transactions are not compatible with other forms of concurrency control such as synchronization variables. In other words, accessing a given data object both inside and outside the scope of an atomic block concurrently leads to undefined behavior. It is the responsibility of the programmer to ensure that shared data accessed inside atomic statements are not accessed outside atomic–statements. This restriction can be relaxed across program phases separated by say barrier operations.

An alternate model for atomicity is the strong atomicity model where transactions are guaranteed to execute in an atomic manner with respect to all operations. Our choice of weak–atomicity is primarily driven by the performance overheads and implementation challenges in supporting strong atomicity. We plan to revisit this aspect of our design as part of future work.

5.1.1.3 Nested Atomic Transactions

One of the primary benefits of atomic–statements over locks is that atomic statements are composable. In other words, atomic block statements can be nested and must follow the scoping rules of block statements. We currently provide flat-nesting semantics for nested atomic–statements. An abort at any nesting depth within the transaction returns the control to the beginning of the outer-most transaction. This choice is primarily motivated by the simplicity of implementation but also by the lack of compelling code examples that require more sophisticated transactional nesting semantics such as closed- and open-nesting semantics.
5.1.1.4 Locality Control

One of the essential aspects of the Chapel language is its support for specifying the placement of data and tasks to optimize for locality using the on statement. The syntax of the on statement is defined as follows:

```
on expression do statement
on expression { statement-block; }
on expression do statement
```

The above code snippets execute the statement or statement-block on the target locale specified by the expression following the on keyword.

In a similar sense, the on statement can be combined with single–locale and multi–locale transactions to optimize for locality. In the case of single–locale transactions, the atomic transaction is created, executed, and committed on the target locale. The target locale may refer to the locale on which the current task is executing or may refer to a remote locale. If the transaction is executing locally then there is no additional remote communication required to execute the transaction. The core computation of the RA kernel captures this case. The following code executes the update to the idx2 entry on the locale that holds the data:

```
on T(idx) do atomic T(idx) ^= r;
on T(idx1) do atomic {
  T(idx1) ^= r;
on T(idx2) atomic T(idx2) ^= s;
}
on T(idx2) atomic T(idx2) ^= s;
```
If any of the operations in the inner `atomic`-statement were to fail, then the whole transaction must abort (given flat–nested transaction semantics). This would mean the transaction rollbacks its speculative state (as defined by the underlying implementation), and control returns to the first statement of the outermost atomic block. The transaction retires until it can complete all its operations without conflicting with any other transaction.

An interesting question is the equivalence between the following two cases:

```plaintext
on x.locale do atomic x++;
atomic on x.locale do x++;
```

The first statement executes the transaction on the target locale identified by `x.locale`. However, the second case creates a transaction locally, executes the `on` statement on the locale referred by `x.locale`, and commits the transaction. This implies the transaction is a multi–locale transaction. An optimized implementation may choose to implement the second statement similar to the first. However, the semantics of the atomic block and the `on` statement does not guarantee such optimizations.

Following the nesting semantics presented earlier, we also allow `on` constructs to be arbitrarily nested within `atomic`-statements.

### 5.1.2 Prohibited Operations within Atomic Statements

In general, we do not permit operations that cannot be executed in isolation, rolled back, or that interfere with the progress guarantees of atomic transactions. Currently, the following operations are prohibited within `atomic`-statements: I/O operations, blocking synchronization operations such as accessing synchronization variables, and calling `extern` functions. A number of proposals support both I/O and lock-based operations within transactions. Whether these features within Chapel’s `atomic`-statements remains
to be seen. Chapel support for calling external C functions is supported via its extern capability. If such a function is called within atomic block then we need to be able to instrument any operations that access shared state.

5.1.3 Advanced Features

The above discussion on the semantics of atomic–statements is by no means complete. Further, we have not attempted to formally define the semantics of atomic–statements here. We describe these in greater detail below.

5.1.3.1 Forward Progress

The forward progress property ensures that threads seeking to modify shared objects will eventually complete their operations (Section 2.3.1). Our current specification guarantees the simplest deadlock-free implementation. The underlying STM implement must additionally provide an implementation free from live-locks as well. We are currently investigating the benefits and applicability of more sophisticated forward progress guarantees like wait-freedom, lock-freedom and obstruction freedom to atomic transactions in Chapel.

5.1.3.2 Atomic Support for Classes

Chapel is a statically typed language. Its rich set of data types include predefined primitive types, enumerated types, locality types, structured types (classes, records, unions, tuples), and data parallel types (ranges, domains, arrays).

Early on we had to choose between a word-based or an object-based STM design for keeping track of transactional accesses. We decided to implement a word-based STM given its flexibility in supporting a wider-range of types than an object-based
STM designs. An object-based STM enforces transactional semantics only at the level of classes.

One of the unique aspects of data-parallel types in Chapel is that it is implemented using Chapel classes. This would imply an object-based STM would in fact provider a wider support. The challenge here would be object-based STMs are implemented by cloning whole objects. However, since arrays can be quite large, cloning entire arrays may be quite inefficient as well as. We are currently investigating an hybrid approach for supporting both object-based and word-based STM.

5.1.3.3 Parallelism within Atomic Statements

Chapel provides a wide range high-level abstractions for data and task parallelism. For example, the programmer can create data-parallel loops using `coforall` and `forall` statements. Unstructured task parallelism can be specified using `begin` and `cobegin` statements. These statements may create new tasks to execute the specified loop body or statement block in parallel. Our current proposal for `atomic`–statements does not allow parallel task creation within a transaction. This requires reasoning about how conflicts get resolved between these parallel tasks that were spawned within a transaction and whether to abort all the tasks when one of them aborts.

5.2 Compiler Support for Atomic Statements

Chapel’s compilation process, is organized as a sequence of passes. The compiler starts with building an intermediate representation (IR), also called the Abstract Syntax Tree (AST), from the input Chapel source code. This representation is modified by subsequent compiler passes finally culmination in the generation of C language code. Adding compiler support for atomic transactions essentially transforms the AST and
uses a STM runtime library to implement its functionality. We primarily modified two existing passes, `parse` and `codegen`, and added a new pass to handle atomic transactions. We describe these in greater detail below.

5.2.1 Building the AST

The `Parse` pass is responsible for building the AST from the Chapel source code. Today the compiler parses the statement or the block of statements following the atomic keyword but ignores the keyword itself. Thus, there is no indication in the AST of the presence of atomic statements in the user code. This results in atomic statements being treated like regular non-transactional statements.

The first change to the compiler involved preserving the information about the atomic keyword in the AST. Thus, as we insert the entire block statement into the AST we mark it with a special tag called `BLOCK_ATOMIC`. As such, the presence of this tag is only recognized in the new compiler pass, namely the `insertTransactions` pass, we added to handle atomic statements. All other compiler passes leading to the `insertTransactions` pass treat such block statements as regular block statements, and apply the necessary transformations and optimizations.

We added a compiler flag, `--ignore-atomic`, which if turned on, reverts to the original behavior. This flag instructs the compiler to ignore all occurrences of the atomic keyword. This flag forces the compiler to process the block statement following the atomic keyword as though the keyword was absent. This flag is useful when compiling the program with the `--serial` flag which serializes all parallel constructs and thus avoids the overhead of STM completely. We describe another practical use for this flag in Section 5.5.
5.2.2 Inserting Transactional Primitives

The insertTransactions pass replaces atomic block with calls to transactional primitives. At the highest level, the insertTransactions pass transforms the AST as follows:

- Inserts transaction begin and commit calls to mark the scope of BLOCK_ATOMIC block statements. Also inserts transaction commit calls within this scope in order handle break, continue, and return statements.
- Replaces non-transactional memory operations that access shared data within the BLOCK_ATOMIC block statements with transactional equivalents.
- Creates a clone of all the functions that are called within the dynamic extend of the BLOCK_ATOMIC block statements and recursively handles all the memory and function calls within the body of cloned functions. Clone copies of functions are cached to avoid creating multiple clones of the same function. We describe the need and mechanism for cloning functions in Section 5.2.2.2.

The insertTransactions pass is one of the last passes to modify the AST in non-trivial ways. The rationale behind this is quite straightforward. First, any pass that follows the insertTransactions pass must deal with new transactional primitives added to the AST. Second, we directly benefit from all the compiler optimizations that occur in the earlier stages of the compilation process. In the current setup, the insertTransactions pass happen right after the insertWideReferences pass, which is responsible for introducing wide-references and wide-classes into the AST.

The STM compiler pass begins with constructing a queue of block statements that have the BLOCK_ATOMIC flag set. We iterate through this queue and process the primitive calls and function calls within each of these block statements. We describe each of these in greater detail in Section 5.2.2.1 and Section 5.2.2.2 respectively.

5.2.2.1 Primitive Calls

Primitive calls are fundamental units or operations that the compiler can reason about. For example, one of the most commonly used primitives is the PRIM_MOVE
primitive. This primitive in its simplest form is used to implement C-level assignments, i.e. direct assignment of RHS expression to a LHS symbol. Now, a remote get operation is represented by a PRIM_MOVE but contains a PRIM_GET_REF primitive call in its RHS. The PRIM_GET_REF primitive requires at the very least two additional arguments: locale and address of the get operation. Thus, this primitive call implements the assignment of a remote memory location to a local variable. During code generation, the compiler translates this primitive call to an appropriate runtime call that implements remote get operations.

When processing a given BLOCK_ATOMIC statement, the insertTransactions pass replaces all non-transactional primitive calls that manipulates memory with transactional equivalents. This covers all non-transactional primitive calls that manipulate memory using wide-references. This is because wide-references always operate on the heap and may potentially modify shared data. In the above example, we replace the PRIM_GET_REF primitive with a PRIM_TX_GET_REF primitive. The new primitive requires an additional argument namely the transaction descriptor.

Primitive calls extend beyond memory operations. For example, we use primitives PRIM_TX_BEGIN and PRIM_TX_COMMIT to represent the transactional begin and commit operations. The insertTransactions pass is also responsible for inserting these primitive calls into the AST at the beginning and end of BLOCK_ATOMIC blocks respectively.

5.2.2.2 Function Calls

As mentioned in Section 5.1, function calls that appear within the dynamic scope of a transaction have to be cloned. This is necessary for two reasons. First, the same function may be called from both inside and outside atomic–statements. Second, we
need to pass the same transaction descriptor for the invoked functions from inside a
given transaction. We create a clone of all the functions that are called from inside atomic–
statements and recursively process all the primitive operations and function calls within
their scope. Unlike top-level block statements that were originally added to the queue, the
entire body of the cloned functions have to be processed.

There are two types of function calls that need to be handled. First are regular
function calls that get called directly by the calling thread. These are handled as follows:

1. We create a clone of the function by making a copy of the function body and
adding a prefix __tx_clone__ to its name.
2. We insert the transaction descriptor argument (described in Section 5.4.1) as the
first formal argument.
3. We add the entire function body into the queue of atomic block statements that
are yet to be processed. This allows us to process all the primitive operations and
function calls within the entire function body.

The second category belongs to functions that are invoked through the RPC mech-
anism. In this case, we need to clone two functions. First, is the actual function that is
being invoked. The second is a routine that is inserted by the compiler. This routine is
essentially a wrapper around the original function. However it is used to marshal data
to and from the original function, and is called via the runtime’s RPC mechanism. Once
we clone these functions, we use transactional RPC mechanisms to invoke the cloned
routines. This mechanism is similar to how GTM invokes its remote procedures.

We keep track of functions that have already been cloned. Each time we encounter
a function call within the atomic block statement we check to see if that function has
already been cloned. If it is cloned then we use the already existing cloned copy other-
wise we create a clone as we just described. We extend the same mechanism to handle
recursive functions based on the fact that the clone of a transactional clone is itself.
5.2.3 Code Generation Pass

The code generation pass, or codegen for short, traverses the AST and generates C code corresponding to the various declarations, primitive calls, function calls in the AST. We rely on existing compiler mechanisms to generate C code for transaction descriptor declarations and function prototypes of transactional clones. The new addition to this pass is the support for generating STM runtime library procedures corresponding to the transactional primitives in the AST. We describe this implementation–independent interface in the next section.

Consider the non-transactional remote get primitive, PRIM_GET_REF, described earlier. The codegen pass generates the following C macro defined by the communication interface:

\[
\text{CHPL\_COMM\_WIDE\_GET}(\text{local, wide, type, ln, fn})
\]

On the other hand, the transaction remote get primitive, i.e. PRIM_TX_GET_REF, is implemented using the interface provided by the STM interface:

\[
\text{CHPL\_STM\_COMM\_WIDE\_GET}(\text{tx, local, wide, type, ln, fn})
\]

Both macros fetch data from the memory location referenced by the wide-reference into the destination location identified by local. The difference however lies in the semantics of these calls; the second macro provides transactional semantics.

5.3 Implementation–Independent STM Interface

The implementation–independent STM interface is an important glue between the compiler and the runtime. Unless otherwise stated, we require any implementation of this interface to be thread–safe and thus be invoked by more than one thread on a given locale. The implementation–independent STM interface consists of two components.
1 void chpl_stm_init();
2 void chpl_stm_exit();
3 chpl_stm_tx_p chpl_stm_tx_create(void);
4 void chpl_stm_tx_destroy(chpl_stm_tx_p tx);
5 void chpl_stm_tx_begin(chpl_stm_tx_p tx);
6 void chpl_stm_tx_commit(chpl_stm_tx_p tx);
7 void chpl_stm_tx_abort(chpl_stm_tx_p tx);

9 chpl_stm_tx_env_p chpl_stm_tx_get_env(chpl_stm_tx_p tx);
10 void chpl_stm_tx_load(chpl_stm_tx_p tx, void* dstaddr, void* srcaddr, size_t size, int ln, chpl_string fn);
11 void chpl_stm_tx_store(chpl_stm_tx_p tx, void* srcaddr, void* dstaddr, size_t size, int ln, chpl_string fn);
12 void chpl_stm_tx_get(chpl_stm_tx_p tx, void* dstaddr, int32_t srclocale, void* srcaddr, size_t size, int ln, chpl_string fn);
13 void chpl_stm_tx_put(chpl_stm_tx_p tx, void* srcaddr, int32_t dstlocale, void* dstaddr, size_t size, int ln, chpl_string fn);
14 void chpl_stm_tx_fork(chpl_stm_tx_p tx, int dstlocale, chpl_fn_int_t fid, void* arg, size_t argsize);
15 void* chpl_stm_tx_malloc(chpl_stm_tx_p tx, size_t number, size_t size, chpl_memDescInt_t description, int32_t ln, chpl_string fn);
16 void chpl_stm_tx_free(chpl_stm_tx_p tx, void* ptr, int32_t ln, chpl_string fn);

Figure 5.1. Implementation-Independent STM Interface
The first component consists of the low-level STM procedures to be implemented by all STM runtime libraries (Figure 5.1). In addition to this, it defines a set of high-level C macros targeted by the Chapel–to–C compiler. These macros builds on the low-level interface to implement high–order features.

In all these methods, the tx argument represents the transaction descriptor and as mentioned earlier is used to store all the private metadata pertaining to a transaction. Further, almost all these interface procedures accept two additional arguments: ln and fn, for diagnostic purposes. These represent the line number and filename where the call is invoked from and is inserted by the insertLineNumbers pass of the compiler as mentioned in Section 5.2.

Now consider the simple example of a transactional load operation. This operation reads the value from the heap and stores it in the transaction’s read set for later use. This functionality is provided by the chpl_stm_tx_load procedure. On the other hand, chpl_stm_tx_get implements a purely–remote transactional load operation. The implementation–independent STM interface provides a high–level macro, called CHPL_STM_COMM_WIDE_GET, to choose the appropriate operation at runtime. We list the implementation of this macro in Figure 5.2. This macro checks if the locale

Figure 5.2. Non-transactional and Transactional Get Operation

```c
#define CHPL_STM_COMM_WIDE_GET(tx, local, wide, type, ln, fn)
1 do {
2     if (chpl_localeID == (wide).locale)
3         chpl_stm_tx_load(tx, &local, (wide).addr,
4             SPECIFY_SIZE(type), ln, fn);
5     else
6         chpl_stm_tx_get(tx, &local, (wide).locale, (wide).addr,
7             SPECIFY_SIZE(type), ln, fn);
8 } while(0)
```
void on_fn82pint21_t _dummy_locale_argu uint64_t r) {
  chpl_stm_tx_env_p local_env;
  chpl_stm_tx_p tx;

  {  
      CHPL_STM_TX_BEGIN(tx, local_env);
      T6 = chpl____tx_clone_dsiAccess(tx, T5, &(this18), 176, "ra.chpl");
      CHPL_STM_COMM_WIDE_GET(tx, T7, T6, uint64_t, 512, "ra.chpl");
      T8 = (T7 ^ myR);
      CHPL_STM_COMM_WIDE_PUT(tx, T6, T8, uint64_t, 512, "ra.chpl");
      CHPL_STM_TX_COMMIT(tx);
  }  
  return;
}

Figure 5.3. HPCC RA Kernel: ATOMIC Version C code

((wide).locale) responsible for the data (pointed to (wide).addr) refers to its own locale or a remote locale. If both the locales are the same, then it performs the transactional load operation else it calls the transactional get operation. The compiler when it is generating code cannot always determine if the references are local or remote.

The code listing in Figure A.1 shows the fragment of the C code that is generated for the HPCC RA kernel (Figure 2.1). Full code can be found in the Appendix A. The purpose of this example is to show that the compiler uses the high-level macros when it generates the code and these macros in turn call the appropriate STM procedure. In all there are around 48 such macros to implement a wide-range of functionality required by the compiler.
5.3.1 Interface Design

We now describe this interface in greater detail. The bootstrapping functionality is provided by the `chpl_stm_init` and `chpl_stm_exit` procedures and implement the initialization and termination functionality respectively. Both these routines are invoked from within the runtime as part of its initialization step, before the application starts to execute the main program. Both these methods will be invoked by a single thread on each locale.

The procedures `chpl_stm_tx_create` and `chpl_stm_tx_destroy` is used to create and destroy transaction descriptors respectively. Both these procedures are purely–locale and do not invoke additional remote communication.

The procedures `chpl_stm_tx_begin` and `chpl_stm_tx_commit` are used to start and commit transactions respectively. The procedure `chpl_stm_tx_abort` can be used to abort transactions. Transactional commit and abort operations may implicitly generate additional remote communication operations.

`chpl_stm_tx_load` is provided to implement purely–local transactional load operations. It performs a transactional load operation of size bytes from `srcaddr` to `dstaddr`, a private memory location (usually on the calling thread’s stack) on the calling locale.

`chpl_stm_tx_store` is provided to implement purely–local transactional store operations. It performs a transactional store operation of size bytes from `srcaddr` to `dstaddr`, a private memory location (usually on the calling thread’s stack) on the calling locale.

`chpl_stm_tx_get` is provided to implement purely–remote transactional load operations. It performs a transactional load operation of size bytes from `srcaddr` on a remote locale (`srclocale`) to `dstaddr`, a private memory location (usually on
the calling thread’s stack) on the calling locale. `chpl_stm_tx_put` to implement
*purely–remote* transactional store operations. It performs a transactional load operation
of size bytes from `srcaddr`, a private memory location (usually on the calling
thread’s stack) on the calling locale, to `dstaddr` on a remote locale (`dstlocale`).

`chpl_stm_tx_malloc` performs a *purely–local* transactional malloc operation,
while `chpl_stm_tx_free` performs a *purely–local* transactional free operation. Both
these procedures are used to manage the global heap storage from inside a transaction.
Memory allocated using `chpl_stm_tx_malloc` is automatically freed on an abort
and memory freed using `chpl_stm_tx_free` is made visible globally only on a
successful commit.

`chpl_stm_tx_fork` implements transactional RPC mechanism. It can be used
to invoke both local and remote transactional procedures. `chpl_txtfetable` holds a
*table of functions that are directly invoked by `chpl_stm_tx_fork`. This table is
generated by the compiler and was described in Section 5.2.3.*

5.3.2 Comparison with GTM Interface

We briefly consider the main differences between the STM interface for Chapel vs
our earlier work on GTM (Section 4.2). First, we do not provide an explicit procedure
for invoking RPC mechanisms from outside a transaction. In other words, we rely on
Chapel’s `chpl_comm_fork` for this purpose. This method is used to invoke pre–
registered procedures that in turn invoke user–level code specified within Chapel’s on
construct.

Chapel also provides a non–blocking variant, namely `chpl_comm_fork_nb`. Second, the transactional malloc and free calls are purely–local in nature. The compi-
ler handles the responsibility of allocating memory appropriately. Finally, the current
interface does not support non-blocking functionality. We plan to explore this as part of future work.

5.4 GTM2: A PGAS Runtime Library for Chapel

There are numerous algorithms and strategies to implement the above STM interface. In this section, we present a prototype implementation of this interface called Global Transactional Memory 2 (GTM2). A number of implementation aspects of GTM2 has been borrowed from GTM (Chapter 4.4). However, GTM2 has been implemented from scratch in order to be tightly integrated into rest of Chapel’s runtime. More importantly, GTM2 uses Chapel runtime’s communication, task, thread, and memory functionality to implements its features instead of relying on a specific implementation. As such, we fully support and have extensively tested GTM2 using the GASNet implementation of the communication layer, FIFO tasks layer implementation, and Pthreads threading layer. We expect to fully support and test other runtime implementation as part of future work.

The core STM algorithm of GTM2 is the same as GTM and is based on a word-based, read versioning, deferred update, and eager acquire scheme. Similar to GTM, we use Hans Boehm’s atomic.ops library [10] to implement memory barriers, single-word atomic swap, and single-word atomic increment/decrement operations. One major change from GTM is that even though we target 64 bit architectures, GTM2 supports non-aligned multiple-word data access operations that are multiples of a single byte. We describe the different components of our implementation below.
5.4.1 Metadata Organization

GTM2 maintains a variety of metadata information for guaranteeing the necessary transactional semantics. This includes: per–node shared metadata, per–transaction private metadata, and per–node shared transaction descriptor table. For the most part, the design and implementation of these metadata structures follow the designs previously described in Section 4.4.1.

In terms of the per–node shared metadata, we allocate a larger table of $2^{24}$ entries per node to help reduce the probability of false conflicts.

In terms of the per–transaction descriptor metadata, we only maintain the following information: transaction’s status, read/write set, nesting level, remote nodes list, free on abort and free on commit lists. We do not maintain any state related to non-blocking operations. GTM2’s shared transaction descriptor table uses a similar algorithm as in GTM.

5.4.2 STM Algorithm

The core STM algorithm consists of two parts: local–sync and remote–sync. The local–sync component implements the STM procedures that are local and is similar to the implementation described in the GTM implementation. The remote–sync component implements STM procedures that are remote. Currently we do not support non-blocking operations in GTM2. For example, `chpl_stm_tx_get` is implemented as follows:

1. Add target node to the transaction’s remote node list.
2. Send a GASNet AM to remote node to invoke get handler.
3. Get handler first locates the transaction descriptor corresponding to the source transaction that issued this request. If this is the first request, it creates a new descriptor and stores it in the shared descriptor array.
4. The handler calls `chpl_stm_tx_load`. If the operation succeeds it returns data to the source node. Else it sends a message to the source node to abort the transaction.

5. The source transaction waits for the target node to send the data. If the data is returned, the transaction continues with its operation and uses the data.

6. If the operation fails, then it aborts the transaction.

The `chpl_stm_tx_put` is similar to the `chpl_stm_tx_get` procedure but calls the corresponding put handler on the target locale.

Unlike our implementation of `gtm_tx_fn` in the GTM prototype, we allow arbitrary nesting of `chpl_stm_tx_fork` methods. In other words, we allow procedures invoked via `chpl_stm_tx_fork` to issue additional `chpl_stm_tx_fork` calls.

The `chpl_stm_tx_fork` procedure is implemented as follows:

1. Add target node to the transaction’s remote node list.
2. Send AM to remote node to invoke fork handler.
3. Fork handler first locates the transaction descriptor corresponding to the parent transaction that issued this request. If this is the first request, it creates a new descriptor and stores in in the shared descriptor array. It then does a `setjmp` to store the state in order to jump back on an abort.
4. The fork handler uses the `fid` to index into the `chpltxftable` and call the appropriate wrapper routine. This routine in turn calls the actual user code corresponding to the `on` construct. When any of the local operations fail, it uses the `longjmp` to jump back to the point of `setjmp` and then return to the source transaction with a failure status.
5. When the source transaction waits for the target locale to return. It continues only if it receives a success. Otherwise, it aborts the transaction if this is the top-level transaction. If not, it in turn reverts to the point of `setjmp` and returns a failure status to its source.

The commit operation is similar to the get operation. It invokes the first phase of commit on all the nodes in remoteprocs. We abort the transaction if any of these operations failed. Next we invoke the second phase of commit on all the nodes. The
second commit phase always succeeds. We added an optimization that allows us to combine both the phases of commit. Earlier, we invoked the first phase of commit on all the nodes and then invoked the second phase of commit on all the nodes once the all first phase commit operations succeed. For transactions that span just two locales, we combine the two remote commit messages into one.

The abort operation waits for all remote requests to complete and then similar to the get operation it invokes the local–sync abort on all nodes in remoteProcs. Once this is done, it re-executes the transaction from the beginning.

5.5 Support Functionality

In the original implementation of Chapel standard/internal modules, Chapel used synchronization variables as its primary concurrency control mechanism to guarantee thread-safety. In most cases synchronization variables acted as locks to guarantee mutual exclusion, while in a few cases they were used directly to additional store some data. As it turns out certain methods in these modules were being invoked indirectly within atomic statements in user-code. However, synchronization variables have blocking semantics and so cannot be allowed inside atomic transactions. To overcome this problem, we replaced a number of synchronization variables and their related synchronization operations with with equivalent atomic transactions. It must be noted that this change is not complete and has been tailored to minimally satisfy the requirements of the sample programs listed in Chapter 6.

The Chapel runtime implements its task report manager using Chapel code and uses a number of standard and internal modules for this purpose. The runtime relies on special compiler flags to make this support possible. For example, it uses the --serial flag to assert that the task report manager code is executed by a single thread.
We additionally added the `--ignore-atomic` flag to this list in order to accommodate the changes we just mentioned to the standard and internal modules to use atomic statements. It is not possible to compile these parts of the runtime without adding this flag because atomic statements are implemented with runtime support but the runtime library requires these Chapel source files to be converted to C code to get build. As such this does not affect performance or correctness since these modules are compiled with the `--serial` flag.

5.6 Summary

This chapter presented the language, compiler, and runtime issues involved in implementing Chapel’s `atomic` statement using PGAS-STM techniques. The next chapter evaluates our implementation using a number of sample benchmarks and application codes.
CHAPTER 6

EVALUATING ATOMIC TRANSACTIONS IN CHAPEL

In this chapter, we evaluate our implementation of atomic transactions in Chapel. We consider the programmability and performance benefits of atomic transactions, and analyze how it compares against lock-based versions of the same kernels. This chapter is organized into three sections. Section 6.1 describes our overall experimental setup. Each of the next two sections, Sections 6.3–6.4, describes a specific workload, compares its atomic version against one or more lock-based versions, and presents performance and scalability results, and provides in-depth analysis of the STM operations performed.

6.1 Experimental Setup

We performed all our experiments on the NCCS Jaguar Cray XT5 system at ORNL. At the time of this writing, this system is rated the world’s fastest supercomputer by the Top500 list of supercomputers [57]. We list some of the relevant hardware and software system configuration parameters in Table 6.1. We list the important Chapel configuration parameters in Table 6.2. Our modifications were performed on a separate branch of the Chapel source repository, and was synced with the main repository on a regular basis. The last merged revision number is r17861 and roughly corresponds
to the Chapel 1.2 release. We use GASNet’s MPI conduit instead of the native Portals conduit on the XT5 system. This is because of a combination of performance and concurrency bugs in using the Portals conduit when used with Chapel. As such, this issue is not related to the modifications we made to the compiler and/or runtime system. It is currently unclear if this issue lies with GASNet’s Portals conduit or how Chapel uses GASNet.
TABLE 6.2

CHAPEL CONFIGURATION PARAMETERS

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Flags/Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapel-to-C compiler flags</td>
<td>–fast –no-devel</td>
</tr>
<tr>
<td>Backend gcc compiler flags</td>
<td>–O3 –target=linux –std=c99</td>
</tr>
<tr>
<td>Chapel Runtime STM Implementation</td>
<td>GTM2</td>
</tr>
<tr>
<td>Chapel Runtime Comm Implementation</td>
<td>GASNet 1.12.0</td>
</tr>
<tr>
<td>GASNet Conduit</td>
<td>MPI</td>
</tr>
<tr>
<td>GASNet Comm Segment</td>
<td>fast</td>
</tr>
</tbody>
</table>

Note, Chapel spawns up to 128 threads per node and reuses it over the execution of the program. It is not as straight-forward to determine the number of threads. However, the lower limit will be 24 given some of our runs use 24 tasks per locale.

6.2 STM Statistics

We collect a wide-range of STM statistics to understand the behavior of atomic transactions in Chapel. We keep track of a number of parameters at a fine-grain per-transaction level. Collecting STM statistics introduces total execution time overheads between 0.1% to 5% compared to when they are not collected. By default, we turn off statistics collection when running the performance experiments. We plan to address the overheads introduced by STM statistics as part of future work.

We collect four different types of STM statistics. Figure 6.1 identifies the different components we track for single–locale (SL) and multi–locale (ML) transactions. As
described earlier, SL transactions execute within the context of the locale on which they were created, while ML transactions access data and store metadata on multiple locales. At the highest-level are metrics that are common to both SL and ML transactions:

**CREATE:** captures the total number and time spent in creating the transaction descriptor. This is a one-time operation and requires the same set of actions irrespective of whether the transaction eventually accesses remote data or not.

**FAIL:** captures the total number and time spent by transactions that have aborted at least once.

**SUCCEED:** captures the total number and time spent when a transaction commits successfully.

Transactions that fail can be further classified into FAIL-SL and FAIL-ML. FAIL-
SL represents transactions that fail without ever having performed any remote operations. On the other hand, FAIL-ML represents transactions that have performed remote operations in one of their executions before they are forced to abort. As such, the actual cause of failure of a FAIL-ML transaction might be a local or remote operation. We further keep track of statistics that of transaction that need to be retried:

**ABORT-SL:** captures the total number and time wasted in failed transactions that are purely-local.

**ABORT-ML:** captures the total number and time wasted in failed transactions that perform remote communication, i.e. ML transactions.

**ROLLBACK:** captures the total number and time spent executing the actual rollback operation. This essentially involves invoking `longjmp` to jump back to the beginning of the transaction. The time also includes the time spent by the transaction as part of the conflict manager is also captured here. For example, this will include the time a transaction waits before it retries its operation.

Next, we keep track of the different operations executed by successful execution of a transaction:

**NON-TX:** captures the total number and time spent inside transactions executing non-transactional operations. This includes both local and remote operations.

**LOCAL:** captures the total number and time spent executing purely-local STM operations.

**COMM:** captures the total number and time spent executing STM operations that require remote communication. This includes the time it takes to setup and send the request to the remote locale.

**HANDLER:** captures the total number and time spent executing requests from remote locales on a given locale. These requests are executed by GASNet handlers. This is complementary to the COMM component and represents how much time a given locale spends executing transactional requests from other locales.

In addition to this, we track the individual operations within the LOCAL, COMM, and HANDLER components as well. The LOCAL component is broken down into
LOAD, STORE, MALLOC, FREE, COMMITPH1, COMMITPH2, and ABORT, representing the total number and time spent by all transactions (on all locales) executing the purely–local transactional load, store, malloc, free, commit phase 1, commit phase 2, abort operations. The abort here represents the time spent in performing the actual abort sequence and not the whole time spent in a failed transaction.

Similarly, the COMM and HANDLER components are broken down into GET, PUT, FORK, COMMITPH1, COMMITPH2, and ABORT, representing the total number and time spent by all transactions (on all locales) executing the purely–remote transactional get, put, fork, commit phase 1, commit phase 2, and abort operations. As in the LOCAL case, the abort operation refers the time spent in executing the actual abort operation and not the whole time.

6.3 HPCC RA kernel

The Chapel code listing in Figure 6.2 shows the overall declarations, setup, and update step of the HPCC RA kernel. We had described the declarations and setup of this benchmark earlier in Section 2.2.2.1. The kernel is divided into two phases. In the first phase the RA kernel performs a random sequence of $NU$ updates on a $m$ element distributed table/array. The benchmark specification requires these updates be done in parallel but does require any concurrency control mechanism to guarantee correctness. We refer to this phase also as the unsynchronized or update step. A second phase reapplies the same sequence of updates as the first phase. We refer to this phase as the verification step. Since the verification step is also done in parallel, the updates have to be performed in a thread-safe manner in order to avoid any additional inconsistencies.

The main change is the loop body itself (lines 18–22). First, the actual expression used to calculate the target locale of the on–statement is changed from $T(indexMask$
const TableDist = new dmap(new Block(boundingBox=[0..m-1])),
TableSpace: domain(1,uint(64)) dmapped TableDist=[0..m-1];
const UDist = new dmap(new Block(boundingBox=[0..NU-1])),
Updates: domain(1,uint(64)) dmapped UDist=[0..NU-1];
var T: [TableSpace] uint(64);
const n = log2(m);

def indexMask() {
  return r >> (64 - n);
}

// Update step -- UNSYNC
const startTime = getCurrentTime();
forall (r, r) in (Updates, RAStream()) do
  on TableDist.idxToLocale(indexMask(r)) {
    const myR = r;
    const myIndex = indexMask(r);
    T(myIndex) ^= myR;
  }
const updateTime = getCurrentTime() - startTime;

// Verification step -- ATOMIC
const start2Time = getCurrentTime();
forall (r, r) in (Updates, RAStream()) do
  on TableDist.idxToLocale(indexMask(r)) {
    const myR = r;
    const myIndex = indexMask(r);
    atomic T(myIndex) ^= myR;
  }
const verifyTime = getCurrentTime() - start2Time;

Figure 6.2. HPCC RA Kernel: UNSYNC and ATOMIC Versions

(r) to TableDist.idxToLocale(indexMask(r)) (line 18). The second expression achieves the same result as the first one. However, it eliminates additional remote communication introduced by the compiler today (unrelated to our work) when attempting to calculate the owning locale directly using the first expression. We expect this to difference to be eliminated as the compiler improves.
Second, a new constant `myR` is declared on the remote locale to capture the value of `r`, implying that the value of `r` will not be changed on the remote locale. This enables an optimization that passes `r` by value to the remote locale, and avoids the additional remote communication required to access `r` from the source locale.

6.3.1 Benchmark Description and Programmability Analysis

We implemented and tested four implementations of the RA kernel (Section 2.2.2), essentially differing in the synchronization mechanism used in the verification step. The `ATOMIC` version (Figure 6.2) uses atomic statements and relies on the compiler to generate the correct STM instrumentation. Here, adding the `atomic` keyword before the loop update operation is sufficient to guarantee each update will execute as an atomic transaction.

The `Sync Data Array` (SDA) implementation declares the table `T` as an array of synchronization variables (Figure 6.17). As explained earlier, the update step (UNSYNC-SDA) must use special functions defined for synchronization variables to access the array in an un-synchronized manner. On the XT5, synchronization variables are implemented using two Pthread condition variables with each requiring 48 bytes of storage. Therefore, we were unable to run the UNSYNC-SDA and SDA versions given the memory requirements of declaring `T` as an array of synchronization variables, especially for large problem sizes used in [17].

The next two lock implementations overcomes this problem by using a smaller array of locks and mapping multiple table entries to a single lock. The `Sync Lock Array` (SLA) implementation uses a separate array of synchronization variables that acts as an array of locks. There are two ways to implement this. One solution is to use Chapel’s `strided domains`. This allows us to change the index set of the lock array without changing how
these indices are mapped to locales. An alternate approach is to declare a new domain map and domain for the lock array. We use the second approach given it used familiar concepts.

This solution introduces three new challenges: First, using a smaller lock array introduces unnecessary serialization of updates that may actually proceed in parallel. This is not a big issue with the HPCC RA kernel given the random nature and large size of the table. Second, for data arrays that do not map evenly across locales, accessing the corresponding lock will may require remote communication impacting the performance in a negative manner. Finally, the code in Figure 6.4 is more complicated than the unsynchronized version in Figure 6.2 or the simple one-to-one mapping of locks to table elements.

The Mutex Lock Array (MLA) implementation (Figure 6.5) is similar to SLA, but uses mutex operations used by the Chapel runtime’s threading layer. On the XT5 sys-
T, TableDist, TableSpace, m -- same as ATOMIC

lk -- input parameter, represents the size of the lock array

const LockDist = new dmap(new Block(boundingBox=[0..lk-1]));
const LockSpace: domain (1, indexType) dmapped LockDist = [0..lk-1];

var Tlock: [LockSpace] sync bool;
const lockMask = m / lk;

// Update step -- same as UNSYNC

// Verification step -- SLA
forall (, r) in (Updates, RandomStream()) {
    on TableDist.idxToLocale(indexMask(r)) {
        const myR = r;
        const myIndex = indexMask(r);
        // calculate lock index based on table index
        const myLock = myIndex >> lockMask;
        const x = Tlock(myLock);
        T(x) ^= r;
        Tlock(myLock) = true;
    }
}

Figure 6.4. HPCC RA Kernel: SLA Version
// LockDist, LockSpace, lk -- same as SLA version
// T, TableDist, TableSpace -- same as SLA version

// Uses the extern interface to access runtime mutex operations

// Change lock/unlock operations

Figure 6.5. HPCC RA Kernel: MLA Version
tem, these locks are directly implemented using Pthread mutexes. We rely Chapel’s ability to invoke externally defined C functions to invoke these methods directly. This would mean that the mutex initialization, lock, and unlock procedures are essentially black-boxes to the compiler. This requires us, the programmer, to manually manage the storage and communication when accessing these mutexes. Thus it is essential to wrap the mutex access inside a `on construct to ensure they are always locally accessed.

6.3.2 Performance and Scalability Analysis

We now present the performance and scalability results for the HPCC RA kernel. We run the entire kernel and time both the update step and the verification step. We then plot the number of locales on the x–axis against the execution time on the y–axis. The UNSYNC and UNSYNC-SDA version refers to the update loop performed without any synchronization, with the array. The ATOMIC, SLA, SDA, and MLA versions use the appropriate concurrency control mechanism as described earlier.

Figure 6.6(a) and Figure 6.6(b) represent the execution time results for two different table sizes, namely $2^{28}$ per locale and $2^{24}$ per locale. Both these perform $2^{18}$ updates per locale. As we increase the number of locales, the size of the table and the total number of updates also increases by the same factor. The large table size, $2^{28}$ entries per locale, matches the problem size used in the Chapel’s HPCC 2009 entry [17]. We now analyze each of the four different implementation in greater detail.

Overall, we observe that on two or more locales, atomic transactions (ATOMIC) perform similar to the unsynchronized case (UNSYNC). This is a significant result and indicates that our STM implementation guarantees correctness without paying a performance penalty or requiring significant changes to the code. The MLA version offers two benefits over SLA and SDA. First, it performs better than both the SLA and
Figure 6.6. HPCC RA Kernel: Effect of Table Size
SDA versions. Second, a single Pthread mutex lock requires just 8 bytes of storage compared to 96 bytes of storage required by a single sync variable. This allowed us to use a lock array which was one half the size of the main table. In other words, two array elements share a single lock.

Figure 6.6(a) and Figure 6.6(b) show that the SLA version has the worst performance of all the kernels tested. We used a lock array 1/8 times the size of the table. In other words, each lock in the lock array was responsible for eight table entries. This was the largest lock array we could allocate for the $2^{28}$ table size. We analyze the effect of the size of the lock array in Section 6.3.4.2.

We were unable to run the UNSYNC-SDA and SDA versions with the larger problem size for reasons explained above. However, there are two results that can be observed for the smaller problem size. First, we can observe that unsynchronized access (UNSYNC–SDA) to synchronization variables is as expensive as synchronized access (SDA). Second, the SDA version performs better than the SLA version for 512 locales or more. This may have to do with the fact that SDA accesses one array and has a better cache.

In general, both the versions that use synchronization variables perform poorly compared to the MLA version of the kernel. That said, synchronization variables offer a much richer set of semantics than Pthread mutex locks and is supported by the compiler.

6.3.3 STM Statistics

We now present some in-depth analysis of atomic transactions in the HPCC RA kernel for the workload presented in Figure 6.6(a).
6.3.3.1 High-Level Statistics

Figure 6.7(a) and Figure 6.7(b) presents the high-level CREATE, FAIL, and SUC-CEED statistics for all transactions. It should be noted that the counts represent cumulative values, i.e. the total number for all transactions on all locales. This is true to all our graphs that represent the count on the y-axis. Figure 6.7(a) shows the weak-scaling nature of our workload. As we increase the number of locales, the number of transactions that commit also increases. Additionally, the characteristics of the workload is such that transactions execute without conflicting with each other.

Figure 6.8 presents the break down of time spent in executing transactions. On a single locale, the time spent creating the transaction descriptor consumes almost 90% of the total execution time. As we increase the number of locales, we still spend a significant portion of the execution time in creating the transaction descriptor.

Creating a transaction descriptor involves two operations. First, we allocate the various data-structures that are used to hold the state of the transaction. The includes the transaction descriptor, read / write set, etc. Second, we use set jmp to save the execution context of the thread before we begin the transaction. Currently, Chapel uses a third-party dynamic memory allocator, namely Doug Lea’s memory allocator, to manage its heap storage. As such, this memory allocator trivially provides thread-safety, i.e. allowing multiple threads to call the memory allocator functions, by using a single lock. We observed this to be one of the significant sources of overhead. Statically allocating these data-structures reduced the total execution time by almost 50% for the single–locale run. The problem with this approach is that we cannot grow the size of the read/write set if the need arises. We are currently exploring alternative dynamic memory allocators that fit the requirements of Chapel programs better. Figure 6.8 presents the break down of time spent when transactions commit successfully. As
Figure 6.7. HPCC RA Kernel: High-level Transaction Statistics
such, one can observe that the majority of time is spent in executing STM operations that are local. We shall now see this in greater detail.

6.3.3.2 Breakdown Local STM calls

Figure 6.10(a) and Figure 6.10(b) present the breakdown of the individual operations in the LOCAL component shown in Figure 6.8. As one can observe, the majority of the operations are local transactional load operations. The next component that takes most time is the commit phase 1. To recall, this step basically validates the read set. This means the version numbers of all read set entries in a transaction’s read set is compared against the version number in the per-node shared metadata array (Section 5.4.1). If the version numbers do not match or the write lock bit is set, then it implies
that some other transaction is currently writing to or has already updated one of the memory locations that maps to the same entry in the metadata array.

6.3.3.3 Read/Write Set Characteristics

The size of the read/write set does not vary with the table size or the number of updates. This is because each transaction always updates one table entry. The average size of the read–set for a single transaction is 168 bytes, implying the transaction is performing 21 8–byte load operations. The average size of the write–set is 8 bytes, implying the transaction is performing a single 8–byte store operation. This is evident if one considers the core operation of the atomic statement:

\[ \text{atomic } T(\text{myindex}) = T(\text{myindex}) \wedge r \]
Figure 6.10. HPCC RA Kernel: Breakdown of LOCAL STM Operations
It is worth considering the reason behind all these load operations for reading a single table entry. As mentioned earlier, distributed arrays in Chapel are implemented using domains and domain maps. When the compiler generates code for accessing distributed arrays it also generates code that refers the domain and domain map of the array. The domain map of the array is what defines the layout of the array in memory. Thus, each transaction performs as many as 20 load operations to determine the exact location of the array corresponding to the actual index. The final load operation performs a transactional load of the value in the memory location corresponding to $T(\text{myIndex})$. Since the store operation immediately follows the load, it reuses this address to perform a transactional store of the new value.

However, as one see this is an overkill for the RA kernel since both these data-structures are declared as constants (line 1 and line 4 in Figure 6.2). Our STM compiler pass does not optimize constant data-structures. This is mainly tied to the fact that the current version of the compiler does not propagate the constant-ness of variables to the later stages of the compilation process. On the flip side, programs that do modify Chapel domains and distributions at runtime will function in a thread-safe manner. For example, this may arise when the index set describing the array either grows or shrinks.

As a simple test, we hand-optimized the generated C code, replacing these additional STM library calls to $T$’s domain and distribution with their non-transactional equivalents. As such this improved the performance only on 1 locale by about 50%. All other results were not affected by this optimization. The reason behind this has to do with the remote communication required by non-transactional parts of the code mask the cost of local STM operations to begin with.
6.3.4 Additional Experiments

In this section we vary runtime parameters such as tasks per locale, size of lock array, etc. to study secondary characteristics.

6.3.4.1 Varying number of tasks per locale

The number of tasks per locale essentially represents the number of tasks used execute the `forall` loop. Increasing this number increases the amount of parallelism. This can be specified for any Chapel program using the built-in command line argument `--dataParTasksPerLocale`.

Figure 6.11(a) – Figure 6.12(b) shows the effect the number of tasks per locale has on the UNSYNC, ATOMIC, MLA, and SLA kernels. Here we test 6, 12, and 24 tasks per locale. We test these for the large problem size, namely $2^{28}$ table entries per locale and $2^{18}$ updates per locale. In general, we observe that the 6 task per locale not only performs poorly but also shows quite a bit of variation in terms of execution time. With 6 tasks per locale, we are not generating sufficient number of requests and are bound by communication latency. This is quite visible in the 1024 locales case. The 12 and 24 tasks per locale perform similar to each other for the most part. We chose 12 tasks per locale for all our experiments since it matched the total number of processor cores per node of the XT5 system.

6.3.4.2 Varying size of Lock Array

The mapping between locks and table entries essentially determines the contention between threads. The worst case scenario is to use one lock per locale. Bigger the lock array, less the possibility of contention. The downside of having a large lock array is two fold. First is the memory footprint of the lock array itself, given each
Figure 6.11. HPCC RA Kernel: Effect of Number of Tasks Per Locale in UNSYNC and ATOMIC
Figure 6.12. HPCC RA Kernel: Effect of Number of Tasks Per Locale in MLA and SLA
synchronization variable requires 98 bytes of storage. Second, given the random nature of the RA kernel, large lock arrays may also contribute to cache misses in addition to the ones introduced by the data table $T$. Figure 6.13 shows the effect of varying the size of the lock array relative to the table $T$ for the SLA version of the kernel. We test three different lock array sizes, $1/8$, $1/16$, and $1/32$ times the table size. We test these for the large problem size, namely $2^{28}$ table entries per locale and $2^{18}$ updates per locale. Thus, the size of the lock array per locale is varied from $2^{25}$ lock entries to $2^{22}$ lock entries and corresponds to m3, m4, and m5 lines in the graph.

The $2^{25}$ lock entries represents the largest lock array we could allocate for the $2^{28}$ table size. This number can also be derived from simple back of the envelope calculations. A single pthread condition variable consumes 48 bytes. Sync variables require
Figure 6.14. HPCC RA Kernel: Effect of Local Block Optimization

two condition variables, amounting to 98 bytes. So a single sync variable requires 12 times the storage of a single table entry (8 bytes or uint(64)). Assuming the array sizes are a power of two, the lock array must be between $8 \times 2^3$ and $16 \times 2^4$ times smaller than the data array. In short, for a data array with $2^8$ entries the lock array must be anywhere between $2^{25}$ and $2^{24}$ entries so as to keep the total storage within or around twice that of a single data array.

In general, we observe the main difference between lock arrays arises only for large number of locales. We chose the SLA-m3 version in all our experiments and refers to the case where each lock in the lock array was responsible for eight table entries.
6.3.4.3 Local Block Optimization

The \texttt{local} keyword in Chapel can be used to indicate that a statement or a statement block does not perform any remote communication. The primary advantage of using the local block within the \texttt{on} construct today is that it enables an optimization that executes the computation in the GASNet handler itself instead of spawning a new task or thread on the remote node. The compiler restricts the set of operations that can be optimized in this manner. For example, atomic statements and synchronization variables are not allowed to execute within the GASNet handler.

In figure 6.2 the \texttt{local}–statement can be used to signal that no remote communication is being performed as part of the array update, \texttt{local T(myindex)^= r}. This condition will always be satisfied because the table update always executes on the locale responsible for that entry. The only version of the RA kernel that permits this optimization to be turned on is the update step that performs no synchronization. Figure 6.14 compares the performance of the UNSYNC (without \texttt{local} keyword) against FFAST (with \texttt{local} keyword). The y-axis represents the execution time of the update step with the \texttt{local}–statement for 6, 12, and 24 tasks per locale and without the \texttt{local}–statement for 12 tasks per locale (same as UNSYNC in Figure 6.6(a))

Ideally, executing the computation within the handler is expected to perform better than spawning or creating a new thread on the remote node. From figure 6.14, we can observe this behavior up to 256 locales. All three FFAST versions, i.e. using 6, 12, and 24 tasks per locale, that run the update operation within the handler performs better than the UNSYNC version that runs the update on a new task it creates on the remote node. Surprisingly, this behavior was not observed on the Jaguar XT4 system when using the same setup. Further investigation is required to determine the exact cause of this behavior.
6.4 RA2 Kernel

The RA2 kernel is a synthetic kernel derived from the HPCC RA kernel. We had first introduced this kernel in Section 2.2.2.2. This kernel performs a sequence of $NU$ updates to two random elements of a $m$ element distributed table/array. The key aspect of this kernel is that transactions access data and maintain metadata state across two locales.

6.4.1 Benchmark Description and Programmability Analysis

We tested four different implementations of this kernel, namely ATOMIC, SDA, SLA, and MLA. The $ATOMIC$ version (Figure 6.15) uses the $atomic$ keyword and relies on the compiler to generate the correct STM instrumentation. We rely on transactional cloning to handle the required instrumentation within the $updateValues$ function (Figure 6.16).

The SDA implementation declares the table $T$ as an array of synchronization variables. As explained earlier, the update step ($UNSYNC$-SDA) must use special functions defined for synchronization variables to access the array in an un-synchronized manner. We had described this version in Section 2.2.2.2 in greater detail. Same as in RA, we implemented SLA and MLA versions to use a separate array of locks. In general, developing a lock-based version of the RA2 kernel is not as straightforward as RA, and has to take into account the following issues:

- Enforce a global order in which locks are acquired. This is required to avoid deadlocks. We acquire the locks in the increasing order of index.
- Release the locks in the order in which they are acquired.
- Ensure the same lock is not acquired twice.
- Implement two versions, one that can be invoked without any synchronization and another that uses locks to protect the critical sections.
// T, TableDist, TableSpace -- same as in RA

def updateValuespmy_9 indexTypeu myS9 indexTyper {
    const myRdx : index{askpmy_u nr0
    const mySdx : index{askpmySu nr0
    if (myRIdx != mySIdx) {
        T(myRIdx) += myRVal;
        on TableDist.idxToLocale(mySIdx) {
            T(mySIdx) += mySVal;
        }
    }
}

forall ( , r, s) in (Updates, RandomStream(seed1), RandomStream(seed2)) do
    on TableDist.idxToLocale(indexMask(r)) {
        const myR = r;
        const myS = s;
        updateValues(myR, myS);
    }

forall ( , r, s) in (Updates, RandomStream(seed1), RandomStream(seed2)) do
    on TableDist.idxToLocale(indexMask(r)) {
        const myR = r;
        const myS = s;
        atomic updateValues(myR, myS);
    }

Figure 6.15. RA2 Kernel: Using atomic statement
```c
void on_fn81pint21_t _dummy_locale_argu uint64_t r,
    uint64_t s) {
    uint64_t myR;
    uint64_t myS;
    chpl_stm_tx_env_p local_env;
    chpl_stm_tx_p tx;
    myR = r;
    myS = s;
    {
        CHPL_STM_TX_BEGIN(tx, local_env);
        chpl_____tx_clone_updateValues(tx, myR, myS);
        CHPL_STM_TX_COMMIT(tx);
    }
    return;
}
```

Figure 6.16. RA2 Kernel: Cloning function calls within atomic statements

```c
// TableDist, TableSpace, m -- same as ATOMIC
var T: [TableSpace] sync uint(64);

// Update Step -- UNSYNC-SDA
forall (, r) in (Updates, RAStream()) do
    on TableDist.idxToLocale(indexMask(r)) {
        const myR = r;
        const myIndex = indexMask(r);
        T(myIndex).writeXF(T(myIndex).readXX() ^ myR);
    }

// Verification Step -- SDA
forall (, r) in (Updates, RandomStream()) {
    const idx = indexMask(r);
    on T(myIndex) do T(myIndex) ^= r;
}
```

Figure 6.17. HPCC RA: Using synchronization variables
6.4.2 Performance and Scalability Analysis

Figure 6.18(a) and Figure 6.18(b) present the weak-scaling results for the RA2 benchmark. We use $2^{23}$ table size and $2^{13}$ updates per locale. We test this for six tasks per locale. In general, we observe the SLA version still performs poorly with respect to SDA and MLA. The primary difference between these experiments is that results presented in Figure 6.18(a) optimize the data-locality of the second transaction using the \textit{on}–statement. As one can observe, using the \textit{on}–statement inside the transaction to optimize for data locality provides better performance.

Figure 6.19 presents the strong scaling results for the RA2 kernel. Locales 1 to 32 run the smaller problem size of $S$ while Locales 64 to 512 run the larger problem size $L$. The smaller problem size $S$ uses a table of size $2^{28}$ total entries and $2^{18}$ total updates. The large problem size $L$ uses a table of size $2^{30}$ total entries and $2^{20}$ total updates. These results use twelve tasks per locale. Unlike the weak-scaling experiment, we do not vary the problem size across locales. We used the optimized version, i.e., the one that uses the \textit{on} construct in all these experiments. In general, we see good strong scaling and weak scaling characteristics for this kernel.

6.4.3 STM Statistics

We now present some in-depth analysis of atomic transactions in the RA2 kernel for the workload shown in Figure 6.18(b).

6.4.3.1 High–level Statistics

Figure 6.20(a) and Figure 6.20(b) represent the total count of committed and aborted transactions for the SL and ML case. It should be noted that the counts represent cumulative values. As such, we observe a large number of transaction aborts (greater than the
Figure 6.18. RA2 Kernel: Weak Scaling Results

(a) Optimizing Data-Locality using on-statement

(b) Remote Memory Access
Figure 6.19. RA2 Kernel: Strong Scaling Results: S/L represents the small and large problem size respectively.
Figure 6.20. RA2 Kernel: Total Count of Committed Transactions and Total Number of Retries
transaction commits) in the 1024 locale case for single–locale transactions. We believe this is a result of livelocks. Live-locks occur when two threads cannot make forward-progress because they repeatedly reach the same point in execution each time but are unable to proceed further. We plan to explore this as part of future work. The important observation that is worth pointing out is that the large number of aborts do not seem to affect the total execution time shown in Figure 6.18(b).

Figure 6.21 presents the break down of time spent in transactions that commit. There are two striking observations that can be made from Figure 6.21. First, on a single locale, the time spent creating the transaction descriptor consumes almost 60\% of the total execution time. This behavior was also observed in RA, however it was far worse in that case, consuming almost 90\% of the total execution time. The second observation one can make about the results is that majority of the time spent in two or
more locales is waiting for remote communication operations to complete. It is interesting to note that the total time nodes spend executing tasks from remote nodes is also negligible. In other words, the nodes are waiting on remote communication operations to complete. This demonstrates the need for supporting asynchronous transactions in Chapel.

6.4.3.2 Breakdown of Local STM operations

Figure 6.22(a) and 6.22(b) present the breakdown of the individual operations in the LOCAL component. As one can observe, the majority of the operations are local transactional load operations. This follows the reasoning we presented for RA. That is, when the compiler generates code for distributed arrays it also generates code that refers the domain and domain map of the array.

6.4.3.3 Breakdown of Remote STM operations

Figure 6.23(a) and 6.23(b) present the breakdown of the individual operations in the COMM component shown in Figure 6.21. Figure 6.24(a) and 6.24(b) present the breakdown of the individual operations in the HANDLER component shown in Figure 6.21. As we mentioned earlier, the HANDLER component complements the COMM component. The sum of remote requests sent by all the locales must equal to the sum of remote requests received by all the locales. This explains why Figure 6.23(a) and Figure 6.24(a) look alike.

However, two things must be noted. First, COMM operations are more expensive since they involve the time it takes to send the request and wait for the reply to arrive. On the other hand, HANDLER operations do not include the communication latency. They measure how long each node spends executing requests from remote nodes. The
Figure 6.22. RA2 Kernel: Breakdown of LOCAL STM Operations
Figure 6.23. RA2 Kernel: Breakdown of COMM STM Operations
Figure 6.24. RA2 Kernel: Breakdown of HANDLER STM Operations
same observation was captured in Figure 6.21 as well, where majority of time was spent executing COMM operations.

6.4.3.4 Read/Write Set Characteristics

The average size of the read–set of a single transaction for the single-locale case is 336 bytes. The average size of the write–set of a single transaction for the single–locale case is 16 bytes. This implies a transaction on an average performs 42 8-byte load operations and two 8-byte store operation. This is twice that of the RA kernel. In the RA2 kernel, we are updating two elements compared to a single element in the case of RA. Now let us consider the RA2 kernel:

```c
on T(idx1) do atomic {  
    T(idx1) ^= r;
    on T(idx2) atomic T(idx2) ^= s;
}
```

In the case of multi–locale transactions, the read set is distributed across two locales. The read-set the transaction maintains on the locale it was created on is 484 bytes. Additionally, it maintains a read-set on the remote locale where T(idx2) is located. The size of the read set on the remote locale is 72 bytes. It must be noted that some of these operations perform 12 byte loads. These operations are essentially reading the value of a wide-reference (4 byte locale id and 8 byte memory reference).

In terms of the write–set, a multi-locale transaction maintains a write set of 8 bytes on the locale it was created on and 8 bytes on remote locale. This is because the transaction is performing two store operations, one locally to T(idx1) and other remotely to T(idx2). As mentioned earlier, the compiler does not have generate code that calculates the address of the store operation as it did with the load operation. It reuses the
address from the load operation. The size of the read/write set does not vary with the problem size. This is because each transaction always updates two table entry as part of each update operation.

6.5 Summary

This chapter presented in-depth evaluation of the support for atomic transactions in Chapel using the RA and RA2 kernel. The results presented in this section can be summarized as follows:

- We presented code examples to show the programmability benefits of atomic transactions over lock-based approaches. This is an important result showcasing the need for language support for atomic transactions.

- We demonstrated good strong and weak scaling characteristics for the problem sizes tested. We showed that our implementation scales up to 2048 nodes with as many as 24 tasks per node.

- We showed how atomic transactions beats, if not simply matches, the performance of lock-based implementations in both these kernels.

- We used detailed STM statistics to identify specific bottlenecks in our current implementation and opportunities for improvement. First, we identified the cost of single locale transactions can be lower by adding support in the compiler to ignore data-structures that are declared as constants.

- We identified two bottlenecks in the underlying system: one with the dynamic memory allocator and another with executing operations directly on GASNet AM handlers, that need to be addressed for better performance and scalability.
CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This chapter is organized as follows: Section 7.1 summarizes the work presented in this dissertation and offers a list of specific contributions. Section 7.2 presents opportunities for future research directions.

7.1 Conclusions

Our work is primarily motivated by the need for concurrency control mechanisms for PGAS programming languages and runtime libraries. In PGAS, threads communicate via a shared heap address space which may in reality be partitioned and distributed across independent memories. Such communication must use concurrency control mechanisms to ensure threads see each other’s updates in a consistent manner. The real challenge in building these mechanisms is that the memory locations that need to be updated may in fact be distributed across different memory partitions. As such, a complete solution must not only approach this problem from a correctness viewpoint, but also address the programmability, performance, and scalability perspectives in order to be relevant to PGAS programming languages and runtime libraries.

This dissertation makes a case for and demonstrates the benefits of using STM schemes to overcome some of the challenges involved in designing complete solutions for large-scale PGAS systems. The work presented in this dissertation addresses the
afore-mentioned problem using a combination of language, compiler, and runtime techniques. First, we provide high-level language constructs that allow the programmer to specify the required concurrency control operations in an intuitive manner. Second, we demonstrate compiler techniques that maps these high-level language constructs to the functionality provided by an underlying STM runtime library. Finally, we developed runtime techniques that not only implements STM algorithms correctly but does so in a scalable manner.

To summarize, our work demonstrates that STM implementations can be scaled to hundreds (if not thousands) of nodes executing tens of thousands of threads, far beyond what has been demonstrated by any other distributed STM implementations. It showcases the programmability advantages of STM schemes over lock-based approaches, while matching, if not exceeding, the performance of lock-based approaches.

7.2 Future Work

This work open numerous opportunities for future research. Our work within the Chapel framework provides us an excellent opportunity to continue to explore the benefits of STM techniques to large-scale parallel applications. One of the valuable near-term contributions of this work would be to make these capabilities available to the wider Chapel user community. There are a number of issues that needs to be resolved to make this possible.

From the language perspective, we believe it is essential to formally define the semantics of the atomic statement. This is especially important for defining the interactions between transactional and non-transactional code segments.

One of the unresolved issues with respect to the compiler today is its support, or lack thereof, for dealing with stack references. Our current solution introduces additional
runtime overhead and is not portable across all communication layer implementations.

In addition to this, there are number of simple compiler optimizations that can be applied in the STM compiler pass. For example, we can choose not to instrument data-structures that are statically declared as constants. This optimization is not implemented today due to a combination of issues, both in our STM compiler pass as well as in the rest of the compiler. Even though our hand-optimized versions of our test workloads showed little or no performance improvement, we believe this optimization is required for correctness and performance reasons in the general case. Finally, we plan to make the compiler support more robust by expanding the use cases to atomic statements beyond what has been currently addressed.

In terms of the runtime implementation, we plan to implement good conflict resolution policies that guarantee against livelocks as well as prioritize remote transactions over local ones. The latter issue is crucial for performance given cost of rollback and retry for remote transactions is more expensive than local ones. In addition, we plan to explore alternate STM algorithms and implementation policies. We plan to explore the possibility of porting open-source STM runtime implementations that are already available. Finally, we plan to reduce the overheads of single-node STM operations to optimize for local transactions.

There are a number of long-term research directions we plan to pursue as well.

First and foremost, we plan to explore the benefits and challenges of implementing non-blocking STM procedures in GTM2 and enable the compiler generate code using such procedures. This additionally requires a formal definition of semantics of asynchronous transactional operations. We believe this shares a number of similarities with the issue of dealing with parallel task creating constructs within the atomic statement.

Next, we plan to explore the possibility of testing our STM implementation on alter-
nate architectures already supported by Chapel (e.g. Cray MTA) or may be supported in a straight-forward manner (e.g. Intel’s Single Chip Cloud Computer).

Third, we plan to expand the language semantics to define interactions between lock-based critical sections and atomic transactions. This would help combine locks with atomic transactions in new ways to benefit from the best of both worlds.

We are currently working with the Chapel language team to resolve both near-term and long-term issues.

In addition the above stated directions, we plan to consider possible extensions to GASNet to provide support for transactional operations. A related opportunity for future work would be to explore possible hardware support in NIC processors to help speedup the performance of global transactions.
APPENDIX A

CHAPEL CODE: HPCC RA KERNEL

The code listing in Figure A.1 shows the generated C code for the HPCC RA Kernel when using **atomic** statements (i.e. ATOMIC version in Section 6.3). The statements in **boldface** represent the declarations, cloned function calls, and macros that implement atomic statements using STM support. This can be compared against the generated C code in Figure A.2 without the **atomic** statement (UNSYNC case).
Figure A.1. HPCC RA Kernel: Generated C code for ATOMIC version
1 void on_fn93(int32_t _dummy_locale_arg, uint64_t r) {
2     uint64_t local_n;
3     uint64_t myR;
4     uint64_t myIndex;
5     uint64_t T2;
6     int32_t T3;
7     BlockArr_uint64_t_l_uint64_t_0 _ret = NULL;
8     int32_t T4;
9     BlockArr_uint64_t_l_uint64_t_0 T5 = NULL;
10     chpl___wide___ref_uint64_t T6 = {0,NULL};
11     uint64_t T7;
12     uint64_t T8;
13     local_n = n;
14     myR = r;
15     T2 = indexMask(r, local_n);
16     myIndex = T2;
17     {
18         T3 = T;
19         this18[0] = myIndex;
20         T4 = T3;
21         T5 = chpl_getPrivatizedClass(T4);
22         T6 = dsiAccess(T5, &(this18), 176, "ra.chpl");
23         CHPL_COMM_WIDE_GET(T7, T6, uint64_t, 512, "ra.chpl");
24         T8 = (T7 ^ myR);
25         CHPL_COMM_WIDE_PUT(T6, T8, uint64_t, 512, "ra.chpl");
26     }
27 }

Figure A.2. HPCC RA Kernel: Generated C code for UNSYNC version
BIBLIOGRAPHY


