Consistency Guarantees for Concurrent Shared Objects: Upper and Lower Bounds

Martha Jane Kosa

March 24, 1995

MARTHA JANE KOSA.

Consistency Guarantees for Concurrent Shared Objects: Upper and Lower Bounds

(Under the direction of Dr. Jennifer L. Welch)

ABSTRACT

This dissertation explores the costs of providing consistency guarantees for concurrent shared objects in distributed computer systems. Concurrent shared objects are useful for interprocess communication. We study two forms of concurrent shared objects: physical shared objects and virtual shared objects.

First we consider physical shared objects called (read/write) registers. Registers are useful for basic interprocess communication and are classified according to their consistency guarantees (safe, regular, atomic) and their capacity. A **one-write algorithm** is an implementation of a multivalued register from binary registers where each write of the simulated multivalued register modifies at most one binary register. A one-write algorithm optimizes writes, speeding up applications where writes outnumber reads. We present the first one-write algorithm for implementing a k-valued regular register from binary regular registers. The same algorithm using binary atomic registers is the first one-write algorithm implementing a k-valued atomic register. The algorithm requires $C(k, 2)$ binary registers. Two improved lower bounds on the number of registers required by one-write regular implementations are given. The first lower bound holds for a restricted class and implies that our algorithm is optimal for this class. The second lower bound, $2k - 1 - \lfloor \log k \rfloor$, holds for a more general class. The lower bounds also hold for two corresponding classes of one-write atomic implementations.

Next we consider virtual shared objects from abstract data types, which are desirable from a software engineering viewpoint. We show that the cost (worst-case time complexity) for operations from abstract data types depends on the amount of synchrony among the processes sharing the objects, the consistency guarantee (sequential consistency, linearizability, hybrid consistency), and algebraic properties of the operations. Sequential consistency and linearizability are equally costly in perfectly synchronized systems under certain assumptions. Hybrid consistency is

not necessarily cheaper than sequential consistency or linearizability. In perfectly synchronized systems, operations satisfying certain algebraic properties cost $\Omega(d)$ (d is the network message delay). To contrast, in systems with only approximately synchronized clocks, for certain classes of abstract data types, sequential consistency is cheaper than linearizability. Our results generalize known results for specific object types (read/write objects, queues, and stacks).

ACKNOWLEDGEMENTS

First, I will recognize all the people who helped me academically during my stay at UNC.

Dr. Jennifer Welch taught the advanced analysis of algorithms class in the spring semester of 1990. This course led to my interest in distributed algorithms. In fact, a homework problem from her class (which I got wrong during the class!) led to my work in shared register implementations. She has been an excellent advisor. She has helped me to improve my technical writing, and she has been a good source of ideas during the (many) times that I have been stuck. Although she (unfortunately) had to leave North Carolina, she did not abandon me. I thank her for giving me the opportunity to work with her during the past couple of years, for supporting me in my attendance at various distributed computing conferences, and for letting me come to Texas and stay at her house in order to work on research with her. I do not think that I could have chosen a better advisor than Dr. Welch.

I had the good fortune to collaborate with Dr. Soma Chaudhuri on part of the work contained in this thesis (shared register implementations). She was instrumental in helping with lower bound proofs because of her astute mathematical insight. She is an excellent role model as a researcher in theoretical computer science.

I wish to thank the other members of my committee, Dr. Donald Stanat, Dr. Kevin Jeffay, and Dr. David Plaisted, for being fair examiners during my exams, and for taking the time to let me explain my work to them. I also wish to thank Dr. Stanat for his very helpful advice during my recent job search. I think that following his advice about presentations in interviews led to my job offers.

I wish to thank Dr. John McHugh for helping me with the dissertation style guides for LaTeX. I wish to thank Dr. Peter Calingaert for his advice and construc-

tive criticism when I taught Computer Science 4 during the summer of 1991. This was very important because my career goal after graduate school is to serve as a faculty member at a college that emphasizes excellence in teaching. I wish to thank Dr. Stephen Weiss for serving as my advisor during my first two years at UNC.

Finally, I would like to thank the Computer Science Department and Graduate School for enabling me to hold fellowships and assistantships so that I could graduate without being in debt.

Now I will recognize the people (and specific dogs) who have provided moral support during my stay at UNC.

I wish to thank my parents, Charles and Jane Kosa, for all their love and support during the past 24 (almost 25) years. They have always encouraged me, never saying that I shouldn't study mathematics or computer science because I am female.

Although they cannot read this, I would like to thank my dog siblings, Buddy and Whitey Kosa, for always being so lovable and friendly. I wish to thank my boyfriend/fiancé Mark Boshart for his friendship, love, and support. He has been extremely patient with me. I also wish to thank my grandmother, Lucille Kosa, for always being so good to write me letters during my time in graduate school. I wish to thank my former roommate, Susan Nagel, for her friendship during my time at UNC. Susan's parents, Bill and Sandy Deibler, have also been kind to me during my time at UNC, inviting me to their home for delicious dinners and cutting and perming my hair, and I appreciate their hospitality very much. I would like to thank Judith Auger (Mark's mom) for her kindness and hospitality during the last couple of years.

Although I do not see them very often, I would like to thank my friends from home in Virginia, Dave Smith, Rose Sorrell, and Ann Claire Mangleburg, for getting together with me from time to time when I was at home.

I would like to thank Michael Dannar, a friend from Granville Towers, for his friendship, going to church with me, and the good-luck present that he gave me before my written comprehensive exam.

Many graduate students in this department have made my stay here more enjoyable, especially Dan Palmer, Jin-Kun Lin, Geoffrey Alexander (officemate), Heng Chu, Injong Rhee, Kevin Denelsbeck, Eileen Kupstas, Mee-Jeong Lee (officemate), Mary Szymkowski (officemate), Chris Ruegger, Greg Bollella, and James Riely.

Many staff members (current and former) in the Computer Science Department have also made my stay here more enjoyable, especially Jeannie Walsh, Wanda Andrews, Donna Boggs, Katrina Coble, Belmon Dean, Carolyn Din, Cathy Hatley, Janet Jones, Fred Jordan, Ernest Parker, Catherine Perry, Sandra Rudd, Natalie Sipes, Monica Waugh, and Kristie Weisner. I would also like to thank Jeannie Walsh for her help during my recent job search. Katrina Coble and Janet Jones have helped me keep abreast of graduate school regulations and guidelines. Donna Boggs saved job advertisements for me.

For the past two years, my home away from home was the Phi Delta Chi house. All the members of Phi Delta Chi Professional Pharmacy Fraternity and current/former residents of the Phi Delta Chi house (not disjoint sets) have made my stay in Chapel Hill more enjoyable, especially Mike Schoen, Mike Little, Jeff Smyre, Donna Wagoner, Valerie Bergman, Carol Lowe (Holmes by now, not married to Kelly), Susan Suddreth, Kelly Holmes, Sonia Shaw, Jeremy and Jester Clemmons (Jester is a good substitute for Buddy and Whitey), Linda Kramer, Nicole Tyner, and Lisa Cameron.

I have mentioned a lot of people, and I'm sure that you're tired of reading this by now. However, the support of all these people (and I apologize if I overlooked anybody) was very important to me and helped me in my pursuit of this degree. Thank you, everybody!

Contents

List of Tables

List of Figures

Chapter 1

Overview

1.1 Background

As the use of computers became more prevalent, people wanted to solve larger and larger problems with computers; thus, system throughput needed to increase. One way to increase throughput is to link computers together in a distributed system (this also includes SIMD parallel processor structures). This is not a far-fetched notion because some autonomous computers already share resources such as printers and disk drives. Also, people discovered (to some degree) how to determine the inherent parallelism of problems. They discovered that parts of problems could be solved in parallel; then the various partial solutions could be combined to obtain a total solution. The structure of the problem can be mapped to the processors in a distributed system. The results of the partial solutions must be integrated. Interprocess communication is the key to coalescing the partial solutions to form a total solution.

Interprocess communication is implemented by using shared memory or message passing. We concentrate on shared memory because its semantics are similar to the semantics provided by a uniprocessor memory and it has been used to solve important synchronization problems (such as mutual exclusion and leader election), thus providing tools for building modular concurrent programs. In a truly concurrent shared memory, reads and writes to the same memory location can occur

simultaneously. What should the users of the shared memory assume about the validity of values returned by their read operations? These validity concerns are expressed in terms of *guarantees*. A guarantee describes possible orders in which memory operations appear to occur. For example, in a shared memory with only one process accessing it (i.e., a uniprocessor memory), a read operation of memory location X will appear to occur after the last write of X that completed before the read started and before any write of X that started after the read completed. For shared memories that can be accessed by multiple processes, many guarantees can be defined. We can classify guarantees by their strengths. A strong guarantee means that there is not much flexibility in the possible orders in which memory operations appear to occur. In our thesis, we determine quantitative measures of the costs of building concurrent shared memory objects which provide various guarantees. Intuition tells us that stronger guarantees may be more costly than weaker ones because the stronger guarantees are more restrictive. However, shared objects with stronger guarantees may simplify programming for the users of the system. We investigate the tradeoff between the strength of a consistency guarantee for shared objects and the cost of providing it.

In order to study the costs of providing consistency guarantees for shared objects, we must first explore how to implement shared objects. We can implement shared objects in two ways: either as physical shared objects or as virtual shared objects. Processes directly access physical shared objects, which can be viewed as *hardware*. If processes do not have a physical shared memory, then they use virtual shared objects to communicate. In order to simulate the shared objects, processes create the illusion of physical shared objects (hence, the term *virtual shared objects*). The processes achieve this simulation by keeping relevant information in their local memories, using message passing to maintain consistency of the objects according to a certain consistency guarantee. Thus, message passing is critical for virtual shared objects. Since we can implement shared objects either as physical shared objects or as virtual shared objects, we must study the costs of both types of implementations.

1.2 Physical Shared Objects

We consider physical shared objects called *registers*, which are memory cells that support concurrent reading and writing by a collection of processes. These registers provide read and write operations to their users. We consider registers because they are the shared memory analog of uniprocessor memory locations. We also believe that they may be relatively easy to implement because of their simple semantics. Although registers cannot be used to solve many complicated coordination problems (i.e., consensus) [Her91], they can be used for basic interprocess communication. Processes are directly connected to the registers via *channels*. We want registers to work correctly without depending on the relative rates at which the accessing processes run; we allow the processes to be totally asynchronous. We also require registers to be *wait-free*, which means that reads and writes of the registers complete in a finite number of steps, regardless of the actions of other processes.

We have previously listed some common features of registers. How can registers be differentiated? One way to classify registers is by the consistency guarantees that they provide to their users. What consistency guarantees can registers provide to their users? Lamport [Lam86] defined the safe, regular, and atomic consistency guarantees for asynchronous shared registers in the presence of concurrent reads and writes. He assumed that writes do not overlap other writes. A read of a **safe** register which overlaps with a write can return any legal value of the register. A read of a **regular** register which overlaps with some writes can return either the value written by the latest preceding write or a value written by one of the overlapping writes. An **atomic** register makes reads and writes appear that they have occurred in a particular order without overlapping, preserving the actual ordering of non-overlapping operations. Every regular register is safe, but a safe register is not necessarily regular. For example, consider a 4-valued safe register with possible values of 0, 1, 2, and 3, where the register initially contains 0 as its value. A read which overlaps with writes of 2 and 3 could return 1, which was never written to the register. 1 would be an invalid value to return if the register were regular. Every atomic register is regular, but a regular register is not necessarily atomic. For example, consider a 3-valued regular register with possible values of 0, 1, and 2, where the register initially contains 0 as its value. Consider two reads R_1 and R_2,

where R_2 begins after R_1 ends. Suppose the reads overlap with writes of 1 and 2. R_1 could return 1, and R_2 could return 2. The pair of values would be invalid if the register were atomic because the reads and writes will not appear to have occurred in a particular order. To R_1, the order of the values written would be $2, 1$. To R_2, the order of the values written would be $1, 2$. We can also classify registers according to the number of possible values of the register (binary or multivalued). Last, but not least, we can classify registers according to the maximum allowable number of concurrent readers of the register (1 or many) and the maximum allowable number of concurrent writers of the register (1 or many). In this work, we only consider registers with multiple concurrent readers and one concurrent writer.

Many researchers have worked on building stronger registers from weaker ones. We will survey this related work later. They have developed many register implementations with varying degrees of complexity. However, we have not seen much work on register implementations which are optimal in some manner (number of registers used, amount of work required by a reader, or amount of work required by the writer).

We now consider register implementations which are optimal in the amount of work required by the writer. A **one-write algorithm** is an implementation of a multivalued register from binary registers where each write of the simulated multivalued register modifies at most one binary register. One-write algorithms are interesting because writes to the registers are fast (optimal), resulting in speedups in applications where writes of shared registers occur more frequently than reads. However, reads must be slow in one-write algorithms, resulting in a tradeoff in time costs [CW90]. Chaudhuri and Welch [CW90] developed the first one-write algorithm implementing a k-valued safe register from $k-1$ binary safe registers. They also proved that at least k binary regular registers were needed in any one-write algorithm implementing a k-valued regular register.

We prove that one-write algorithms for implementing regular (respectively, atomic) multivalued registers from regular (respectively, atomic) binary registers do exist. We show that they are expensive with respect to space (improving the lower bound proved by Chaudhuri and Welch [CW90]). One-write algorithms are expensive with respect to space because they require $\Omega(k)$ binary registers, while

the standard binary encoding of k values only needs $O(\log k)$ bits.

We have the following results:

- the first one-write algorithm implementing a k-valued regular register from binary regular registers. It uses $C(k,2)$ binary regular registers.
- the first one-write algorithm implementing a k-valued atomic register from binary atomic registers. It uses $C(k,2)$ binary atomic registers.
- some algorithm transformation techniques which may be of independent interest:
  - a transformation technique for converting a one-write regular algorithm into a normal form algorithm (one in which readers do not write to shared registers, every reader executes the same protocol, and every reader starts in the same state at the beginning of each read).
  - a transformation technique for converting a symmetric algorithm (in which any sequence S of writes, followed by a write of value a, followed by a write of value b, followed by a write of value a, leads to the same states of the binary registers as S followed by a write of value a, for arbitrary possible values of the multivalued register a and b) into a symmetric algorithm in which readers read registers at most once.
  - a transformation technique for converting a one-write algorithm into a one-write algorithm for a smaller value set using fewer binary registers.
- improved lower bounds on the number of binary registers required by regular and normal form atomic one-write algorithms implementing k-valued registers (the previous lower bound was k binary registers):
  - $C(k,2)$ registers are necessary for toggle algorithms (algorithms such that for each pair of distinct values v and w in the domain of the multivalued register, there is a fixed binary register that is changed whenever the value of the multivalued register is changed from v to w or vice versa).

  - at least $2k - \lceil \log k \rceil - 1$ registers are necessary for symmetric algorithms when $k \geq 4$.

- Showing that all regular one-write algorithms implementing k-valued registers need to satisfy the toggle property would be desirable because our one-write algorithm would be optimal. However, we have found a regular one-write algorithm implementing a 3-valued register that does not satisfy the toggle property (it does satisfy the symmetric property). If all one-write algorithms implementing k-valued registers needed to satisfy the symmetric property, then our lower bounds above would be reasonably close. However, we have found a regular one-write algorithm implementing a 3-valued register that does not satisfy the symmetric property. Both counterexample algorithms use 3 binary registers, matching the lower bound of k registers proved by [CW90].

1.3 Virtual Shared Objects

We study implementations of virtual shared objects. These objects are defined as arbitrary abstract data types. Providing objects from arbitrary abstract data types is desirable from a software engineering viewpoint. Our goal is to quantify the costs of implementations of virtual shared objects. The cost measure that we study is the worst-case time complexity of implementations, focusing on the time required by each operation of the abstract data type. We show that several factors influence these costs.

One factor that can influence the costs of implementing virtual shared objects is the amount of synchrony among the processes sharing the objects. We assume that each node in the distributed system has a clock that runs at the same rate as real time and is accessible to all processes running on the node. We consider two possible variations in the amount of synchrony. We assume that each message sent has a delay in the interval $[d - u, d]$, where u is the uncertainty (if $u = 0$, then we have constant message delay). One possibility is for all clocks in the system to be perfectly synchronized[1]. This is very desirable, but is hard to ensure in practice.

[1]Perfect synchrony is an equivalent assumption to constant message delay, as will be explained

Thus, we also consider a more realistic assumption about the amount of synchrony. This is the case when the clocks in the system are only approximately synchronized. This means that they run at the same rate as real time but are not synchronized initially. Although ensuring perfect synchrony is hard, it is useful to study the inherent costs of implementing virtual shared objects in systems with perfectly synchronized clocks because we automatically obtain the minimum lower bounds on the costs of implementing virtual shared objects in more realistic systems.

Another factor that can influence the costs of implementing virtual shared objects is the consistency guarantee to be provided. We now give some examples of consistency guarantees (*strong*, *weak*, and in between or *hybrid*). In our study, we concentrate on the strong and hybrid guarantees.

We now discuss two strong consistency guarantees: sequential consistency and linearizability. In a **sequentially consistent** system, all operations appear to execute in an order which agrees with the order in which the operations of each process were executed. For each object, this order (for all the operations) is legal according to the semantics of that object's abstract data type. Although the intraprocess ordering of operations is preserved by sequential consistency, there are no constraints on the relative ordering of operations that are performed by different processes. Sequential consistency is a well-studied consistency guarantee; several implementations of distributed shared memory (read/write objects) [ABM93, AW91] provide sequential consistency. A **linearizable** system is a sequentially consistent system with the additional constraint that if op_1 completes before op_2 begins in real time, then op_1 precedes op_2 in the ordering of all operations. Thus, linearizability implies sequential consistency; but the converse does not hold. Linearizability is equivalent to the atomic property for shared registers. It was proposed as a consistency guarantee for implementations of general shared objects by Herlihy and Wing [HW90]. They argue that assuming linearizable objects aids in formal verification of concurrent programs. Linearizability satisfies the locality property; a collection of separate linearizable objects is guaranteed to be linearizable. Thus, a linearizable shared memory can be designed incrementally, one object at a time.

later. Messages can be timestamped; they will not be handled until at least time d after they are sent.

We now discuss weak consistency. In a **weakly consistent** system [LS88], for each process, all operations appear to execute in a way that is consistent with the view of the process. This does not mean that all processes observe the same ordering of operations. Weak consistency for read/write objects is very cheap to implement. Each process performs operations on its local copies of objects, sending messages to all other processes when a write is performed. When an update message for a shared object is received by a process, the process changes its local copy accordingly. However, weak consistency is computationally weak; Attiya and Friedman [AF92] proved that the non-cooperative mutual exclusion problem (where processes not trying to enter the critical section do not participate in the mutual exclusion protocol) cannot be solved by only using weakly consistent read/write objects.

We now discuss a hybrid consistency guarantee. Providing a hybrid guarantee may be less expensive than providing a strong guarantee but may still be powerful enough to be useful in solving nontrivial problems. In the **hybrid consistency** condition defined by Attiya and Friedman, each operation has a weak version and a strong version. Their architecture-independent definition of hybrid consistency generalized consistency guarantees, such as weak ordering [AH90, DSB88] and release consistency [GLL+90, GMG91], that were proposed by the computer architecture community. For the shared memory implementations providing these other consistency guarantees, objects were classified as strong or weak [DSB88], and users could be sure that the guarantees were provided only if their application programs obeyed certain synchronization constraints. Hybrid consistency does not assume these restrictions. Weak and strong operations can be performed on the same object. Strong operations must appear to execute in the same order at all processes, and for any two operations executed by process i, where one is strong, every process observes the operations in the same order as process i. [AF92] define two types of hybrid consistency: one where strong operations are linearizable, and one where strong operations are sequentially consistent. They show that hybrid consistent implementations of shared read/write objects can be more efficient than strongly consistent implementations when mostly weak operations are executed. They also show that hybrid consistent implementations of read/write objects can be used in a non-cooperative algorithm for mutual exclusion and that more weakly

consistent implementations cannot be used in a non-cooperative algorithm for mutual exclusion. Attiya, Chaudhuri, Friedman, and Welch [ACFW93] show that certain programming strategies make it possible to transform programs assuming sequentially consistent read/write objects into correct programs only using hybrid consistent read/write objects. Thus, hybrid consistency is an interesting consistency guarantee to study.

These different consistency guarantees (with the exception of linearizability) are incomparable to the safe and regular consistency guarantees for registers discussed earlier.

Last, but not least, algebraic properties of the operations of abstract data types influence the costs of implementing virtual shared objects. For an abstract data type, we build a commutativity graph which represents relationships among the operations. The nodes of the commutativity graph are the names of the operations of the abstract data type. Nodes are connected by an edge if instances of them (with arguments and return values) do not commute. We now explain what it means for two operation instances not to commute. Two operation instances op_1 and op_2 **immediately do not commute** if there exists a sequence of operation instances α such that each of op_1 and op_2 can legally follow it (according to the semantics of the abstract data types being implemented), but at least one of $\alpha \cdot op_1 \cdot op_2$ and $\alpha \cdot op_2 \cdot op_1$ is not legal. In addition, op_1 and op_2 are **cyclically dependent** if both of $\alpha \cdot op_1 \cdot op_2$ and $\alpha \cdot op_2 \cdot op_1$ are illegal. If op_1 and op_2 have the same operation name and immediately do not commute, then we say that op_1 **immediately does not self-commute**. If op_1 immediately does not self-commute, then there is a loop (op_1, op_1) in the commutativity graph. The structure of the commutativity graph helps us to determine the costs of implementing virtual shared objects.

We determine lower and upper bounds on the time complexity of implementing virtual shared objects from general abstract data types. We vary the amount of synchrony among the processes and the consistency guarantee to be provided in order to determine their effect on the time complexity of implementing an abstract data type. The algebraic properties of the operations interact with the previously mentioned parameters.

We first consider systems with perfectly synchronized clocks. We formally prove

lower bounds on the amount of time required for single operations, pairs of operations, and all operations for a type. We have proofs of these lower bounds for both sequential consistency (a lower bound for sequential consistency implies lower bounds for linearizability) and hybrid consistency. With these lower bounds, we have linearizable (and also sequentially consistent) implementations of classes of abstract data types with time complexities that asymptotically match the lower bounds.

We also consider systems with only approximately synchronized clocks. We formally prove lower bounds on the amount of time required for single operations in linearizable implementations. These bounds are higher than the upper bounds holding in systems with perfectly synchronized clocks. We provide sequentially consistent implementations of classes of abstract data types with time complexities that asymptotically match the lower bounds for systems with perfectly synchronized clocks (which also hold for systems with only approximately synchronized clocks).

What conclusions can we draw from our study of the costs of implementing virtual shared objects from general abstract data types? The following are true under certain reasonable assumptions to be explained later (the assumptions are not necessarily the same for each case):

- Sequential consistency and linearizability are equally costly in systems with perfectly synchronized clocks.
- Linearizable operations are more expensive in systems with only approximately synchronized clocks than in systems with perfectly synchronized clocks.
- In systems with only approximately synchronized clocks, sequentially consistent operations are cheaper than linearizable operations.
- Depending on the data type to be implemented, hybrid consistency is not necessarily cheaper than providing a strong consistency guarantee, even when weak operations are used mostly.

Our conclusions generalize known results ([AW91, MR92, AF92]) for specific data types (read/write objects, queues, and stacks). Our conclusions also provide

new information about other types, such as read-modify-write objects, test-and-set objects, dictionary sets, and cyclic arrays[2].

We now list our results in detail.

1.3.1 Systems with Perfectly Synchronized Clocks

1.3.1.1 Sequential Consistency and Linearizability

We present the lower bound results first. These lower bounds hold for sequentially consistent implementations of shared objects. Since linearizability implies sequential consistency, these lower bounds also hold for linearizable implementations of shared objects. These lower bounds depend on the network message delay, which is d.

- The total time for a pair of operations that immediately does not commute is at least d.
- An operation which immediately does not self-commute must take at least time d.
- An operation which is cyclically dependent with other operations must take at least time d.
- The total time for n operations forming a clique in their commutativity graph is at least $nd/2$.
- The total time for n operations forming a clique in their commutativity graph with s of them having to be slow (take at least time d) is at least $sd+(n-s)d/2$.
- The total time for a set of operations is at least $(s+m)d$, where s is the number of operations that must be slow and m is the size of the maximum independent edge set in the commutativity graph formed by removing the slow operations and their incident edges.

[2]Cyclic arrays are arrays in which some operation reads component i and writes to component j and another operation reads component j and writes to component i.

We can use these lower bounds to obtain lower bound results for specific abstract data types. We give a few examples here. For read/write objects, the total time for a read and a write must be at least d (read and write immediately do not commute). For queues, a dequeue operation must take at least time d (dequeue immediately does not self-commute). A read/modify/write operation must take at least time d (read/modify/write immediately does not self-commute). For cyclic arrays with two components where one operation reads the first component and writes the second component and another operation reads the second component and writes the first component, the total time for both operations must be at least $2d$ (the operations are cyclically dependent). For set objects with find, insert, and update operations, the total time for the three operations must be at least $3d/2$ (there is a 3-clique in the commutativity graph).

Now we present the upper bound results. These upper bounds are for linearizable (implying sequentially consistent) implementations of shared objects.

- an implementation of an abstract data type with operation op where $|op| = 0$ (meaning only local computation time, much less than the network delay, is required) if op immediately commutes with itself and is self-oblivious (the return value of an instance of op does not depend on the interactions among any previous instances of op and other operations).
- an implementation of an abstract data type with n operations, such that its commutativity graph is the complete graph on n nodes, with the sum of the worst-case completion times for all n operations equal to $nd/2$.

1.3.1.2 Hybrid Consistency

We have the following lower bound results:

- The total time for a pair of operations (one strong, one weak) that immediately does not commute is at least d.
- Either version (strong or weak) of an operation which immediately does not commute with itself must take at least time d.

- The total time for a pair of cyclically dependent operations is at least $2d$.
- The total time for n operations (both strong and weak versions) forming a clique in their commutativity graph is at least
  - $(n-1)d$ if n is odd.
  - nd if n is even.
- The total time for n operations (both strong and weak versions) forming a clique in their commutativity graph with s of them having to be slow is at least
  - $2sd + (n-s)d$ if $n-s$ is even.
  - $2sd + (n-s-1)d$ if $n-s-1$ is odd.
- The total time for a set of operations (both strong and weak versions) is at least $2(s+m)d$, where s is the number of operations that must be slow and m is the size of the maximum independent edge set in the commutativity graph formed by removing the slow operations and their incident edges.
- $|Waop| + |Wop_1| \geq d$ (Wop means the weak version of op; aop is an accessor, an operation which does not change its object.) or $|Waop| + |Wop_2| \geq d$ if aop returns different results in executions where op_1 and op_2 immediately precede it, respectively.

As with the lower bounds for sequential consistency, we can use these lower bounds to obtain lower bound results for hybrid consistent implementations of specific abstract data types, remembering to use weak or strong versions of operations when appropriate.

1.3.2 Systems with Approximately Synchronized Clocks

In systems with approximately synchronized clocks, the uncertainty in the message delay, denoted by u, plays an important role in our lower bounds.

1.3.2.1 Linearizability

We present some lower bound results:

- A modifier operation (informally, an operation which changes the state of an object) must take at least time $u/2$ (u is the uncertainty in the message delay) if there exists an accessor (informally, an operation which returns some information about the state of an object but does not change it) which can distinguish the order in which two instances of the modifier were executed.
- A modifier operation must take at least time $u/2$ if there exists an accessor which can tell whether one or two instances of the modifier were executed.
- An accessor operation must take at least time $u/2$ if there exists a modifier such that the accessor can tell whether an instance of the modifier was executed.

We can use these lower bounds to obtain lower bound results for specific abstract data types. We give a few examples here. For read/write objects, the time for a read (respectively, write) operation must be at least $u/2$. For queues, an enqueue operation must take at least time $u/2$. For cyclic arrays, each operation must take at least time $u/2$. For set objects with find, insert, and update operations, the find and update operations must take at least time $u/2$.

1.3.2.2 Sequential Consistency

We discuss our upper bound results for sequential consistency.

- an implementation of an arbitrary abstract data type with accessor operation op in which $|op| = 0$.
- an implementation of an arbitrary abstract data type with pure modifier operation (informally, a modifier which does not return any information about an object's state) op in which $|op| = 0$.

These implementations are very similar to the implementations in systems with perfectly synchronized clocks. An atomic broadcast algorithm is used to ensure consistent order of message delivery. All "slow" operations take time h, where h is the time for message delivery by the atomic broadcast algorithm.

Instantiating these implementations yields implementations for specific abstract data types. We give a few examples here.

- an implementation of read/write objects where $|READ| = 0$ (accessor).
- an implementation of read/write objects where $|WRITE| = 0$ (pure modifier).
- an implementation of queues where $|ENQUEUE| = 0$ (pure modifier).
- an implementation of set objects where $|FIND| = 0$ (accessor).

1.4 Contributions

Our study of physical shared objects shows that the classic "space-time tradeoff" holds for one-write register implementations. We can make writes inexpensive (optimal) with respect to time. However, these implementations need a lot of space and reads must take a lot of time.

Although our bounds on space are not tight in all cases, we have determined some transformation techniques which may be of general interest. Perhaps these transformation techniques could be used in improving the lower bounds.

Our results about physical shared objects are interesting because they add rungs to the "ladder", the complexity hierarchy of concurrent registers. They also show how combinatorial techniques can be applied in the design and analysis of asynchronous distributed algorithms.

Our study of virtual shared objects helps us to gain a better understanding of the costs of implementing distributed virtual shared objects. Our results tell programmers that they cannot implement shared abstract data types whose operations complete in time less than our lower bounds. These lower bounds use

algebraic properties of the data types. In some sense, these results reinforce the notion that strong consistency is costly to provide. Hybrid consistency can be as costly to provide as strong consistency, depending on the type of objects to be implemented.

In this thesis, we explore the costs of providing various consistency guarantees for both physical and virtual shared objects. What are the relationships between these two ways of implementing concurrent shared objects? We have formally proved a cost tradeoff that holds for both physical and virtual shared objects under all assumptions about the amount of synchrony among the processes sharing the objects. The tradeoff is as follows: if one operation is fast, then an operation not commuting with it must be slow. In the case of physical shared registers, we have concentrated on implementations where write operations are fast, forcing read operations to be slow. We have also provided a formal reinforcement of the intuition that strong consistency is expensive to implement for both physical and virtual shared objects. For physical shared registers with fast write operations, we have shown that strong consistency is expensive in terms of space as well as time.

Chapter 2

Registers

2.1 Introduction

In any concurrent system, processes need to communicate with other processes. Concurrent reads and writes of shared memory cells, or registers, are required for communication. If the shared memory provides stronger consistency guarantees, then it is more useful to the users of the system, but implementing the shared memory may be more difficult. Thus it is helpful to know which types of registers can implement which other types and what the costs of these implementations are. Many such implementations have been developed, for example, [Blo87, BP87, CW90, Lam86, LTV89, NW87, Pet83, SAG87, Tro89, VA86, Vid88].

In this part of our thesis we focus on implementing a k-ary regular (respectively, atomic) register, the *logical* register, out of binary regular (respectively, atomic) registers, the *physical* registers, for $k > 2$. We assume that our registers support multiple concurrent readers but only one concurrent writer. A *k-ary* register can take on k different values; *binary* means 2-ary. The term "regular" refers to the consistency guarantee provided in the presence of overlapping reads and writes: a read of a *regular* register must return either the value of the most recent preceding write (a well-defined notion since there is only one writer) or the value of an overlapping write. The term "atomic" refers to a stronger consistency guarantee provided in the presence of overlapping reads and writes. For an *atomic* register, the values