imply that every call eventually has a matching response and that call events and response events alternate at a given process (a process never has more than one pending call). If s is a sequence of operations and p is a process, then we denote the restriction of s to operations of process p by $s|p$.

Sequential consistency ensures that all processes agree on some ordering of the execution at the granularity of entire operations. In this ordering, the operations for each process appear in the order in which they were executed at that process.

Definition 1 *An execution ρ is* **sequentially consistent** *if there exists a legal sequence τ of operations such that τ is a permutation of $ops(\rho)$ and $ops(\rho)|p = \tau|p$ for each process p.*

Like sequential consistency, linearizability ensures that all processes agree on some ordering of the execution (at the granularity of entire operations) that preserves the operation sequences of the processes. In addition, the ordering must preserve the actual timings of operations.

Definition 2 *An execution ρ is* **linearizable** *if there exists a legal sequence τ of operations such that τ is a permutation of $ops(\rho)$, $ops(\rho)|p = \tau|p$ for each process p, and op_1 precedes op_2 in τ if the response for op_1 precedes the call for op_2 in $ops(\rho)$.*

Hybrid consistency tries to give us the best of two worlds: strong consistency (in which operations appear to execute everywhere in some fixed order – i.e., sequential consistency or linearizability) and weak consistency (in which operations appear to execute in different orders at different processes, admitting fast implementations). Each operation has a strong version and a weak version. We want all processes to perceive

1. the same order of execution for all strong operations.

2. the same relative order of execution for each pair of operations executed by the same process, where at least one operation in the pair is strong

Definition 3 *An execution ρ is* **hybrid consistent** *if there exists a serialization σ of the strong operations of ρ such that for each process p, there exists a legal sequence τ_p of operations such that the following are true:*

- *τ_p is a permutation of $ops(\rho)$.*
- *if op_1 and op_2 are executed by the same process, op_1 precedes op_2 in ρ, and at least one of op_1 and op_2 is a strong operation, then op_1 precedes op_2 in τ_p .*
- *if op_1 precedes op_2 in σ and both are strong, op_1 precedes op_2 in τ_p.*
- *$\tau_p|p = \rho|p$.*

We now give some examples of hybrid consistent, sequentially consistent, and linearizable executions. We assume that the initial values of X and Y are 0 in all examples.

1. **A Hybrid Consistent Execution ρ_1.** Let $ops(\rho_1)|p_1 = Wwrite(Y,1) \cdot Wread(X,0)$, and let $ops(\rho_1)|p_2 = Wwrite(X,1) \cdot Wread(Y,0)$. All operations in ρ_1 are weak. ρ_1 is hybrid consistent because we can find a serialization of the strong operations of ρ_1 (an empty sequence since there are no strong operations) and legal operation sequences τ_1 and τ_2 satisfying the requirements of Definition 3. Let $\tau_1 = Wread(Y,0) \cdot Wwrite(Y,1) \cdot Wread(X,0) \cdot Wwrite(X,1)$, and let $\tau_2 = Wread(X,0) \cdot Wwrite(X,1) \cdot Wread(Y,0) \cdot Wwrite(Y,1)$. If all operations in ρ_1 were strong instead of weak, ρ_1 would not be sequentially consistent, and thus would not be linearizable, because p_1 and p_2 do not see the same ordering of the write operations. To p_1, the write of Y appears to occur before the write of X. To p_2, the write of X appears to occur before the write of Y.

2. **A Sequentially Consistent Execution ρ_2.** Let $ops(\rho_2)|p_1 = write(X,5)$, $ops(\rho_2)|p_2 = read(X,5)$, and $ops(\rho_2)|p_3 = read(X,0)$. Also, p_1's operation precedes p_2's operation in $ops(\rho_2)$, and p_2's operation precedes p_3's operation in $ops(\rho_2)$. ρ_2 is sequentially consistent because we can legally serialize the operations as follows: $read(X,0) \cdot write(X,5) \cdot read(X,5)$. ρ_2 is not linearizable

because p_3's operation must precede the others for legality, but p_3's operation follows the others in $ops(\rho_2)$.

3. **A Linearizable Execution ρ_3.** Let $ops(\rho_3)|p_1 = write(X,5)$, $ops(\rho_3)|p_2 = read(X,5)$, and $ops(\rho_3)|p_3 = read(X,0)$. Also, p_2's operation starts before p_1's operation finishes, and p_3's operation starts before p_2's operation finishes and before p_1's operation finishes. ρ_3 is linearizable because we can legally linearize the operations as follows: $read(X,0) \cdot write(X,5) \cdot read(X,5)$.

An mcs is a sequentially consistent (respectively, linearizable or hybrid consistent) implementation of a set of objects if any admissible execution of the mcs is sequentially consistent (respectively, linearizable or hybrid consistent).

We measure the efficiency of an implementation by the worst-case response time for any operation on any object in the set. Let O be an object, and OP be a generic operation. $|OP(O)|$ is the maximum time taken by an op operation on O in any admissible execution. $|OP|$, the worst-case time for the generic operation OP to be completed, is the maximum of $|OP(O)|$ over all objects implemented by the mcs.

In our lower bound proofs, we make the argument lists and return lists for operations explicit only when absolutely necessary.

Chapter 8

Strong Guarantees in a System with Perfectly Synchronized Clocks

In this chapter we assume that all processes have perfectly synchronized (perfect) clocks and a constant, known message delay (these two concepts are equivalent[1]). We model perfect clocks by letting the clock functions $C_p(t) = t$ for all processes p and real times t. We model constant message delay by letting the message uncertainty $u = 0$. The mcs processes know the value of the message delay d.

In this chapter we give lower and upper bounds on the costs of providing sequentially consistent and linearizable implementations of virtual shared objects. These lower bounds automatically hold under weaker, more realistic assumptions about process synchrony. We prove our lower bounds on the costs of operations in sequentially consistent implementations, getting lower bounds on the costs of operations in linearizable implementations for free because sequential consistency is weaker. We exhibit four algebraic properties of operations causing individual operations or

[1]If clocks are not perfectly synchronized, but run at the same rate as real time, and the message delay is constant and known, the the clocks can easily be synchronized. On the other hand, if clocks are perfectly synchronized and there is a known upper bound of d on the message delay, then we can simulate constant message delay by timestamping each message sent and having the receivers delay processing of messages until time d after they were sent.

groups of operations satisfying one of those properties to have a worst-case response time of $\Omega(d)$. We demonstrate linearizable implementations of classes of abstract data types in which the worst-case response time for one operation or a group of operations is optimized (the worst-case response time is 0), provided that the operation or group satisfies one of three algebraic properties. If these two lists of properties "cancelled each other out", then we could say that sequential consistency and linearizability are equally costly. Unfortunately, our two lists of properties do not form a predicate and its negation. However, if we are able to optimize an operation or group of operations known to be invoked most frequently, we can say that sequential consistency and linearizability are equally costly for these classes of abstract data types. In addition, if all operations of certain classes of abstract data types are known to be invoked with approximately the same frequency, we can say that sequential consistency and linearizability are equally costly for these classes of abstract data types. This is true because there exist linearizable implementations of the abstract data types with the total worst-case completion time of all operations matching the lower bound on the worst-case completion time of all operations in any sequentially consistent implementation of the types.

Section 8.1 contains our lower bounds on the costs of implementing single operations, pairs of operations, and trios of operations from abstract data types. Section 8.2 contains implementations of classes of abstract data types in which a single operation is optimized. Section 8.3 contains our lower bounds on the costs of implementing all operations from abstract data types. Section 8.4 describes abstract data types for which the bounds in Section 8.3 are tight.

8.1 Lower Bounds for Singles, Pairs, and Trios

In this section we prove lower bounds on the costs of single operations, pairs of operations, and trios of operations in sequentially consistent implementations of abstract data types. These bounds hold if the operations satisfy certain algebraic properties.

We present these algebraic properties as they are needed. These algebraic properties will also be important in Chapter 10, when we consider hybrid consistency.

We now describe algebraic properties which will be important in proving lower bounds on the worst-case response times for single operations and pairs of operations.

Definition 4 *[Wei93] If β and γ are operation sequences, then β and γ* **commute**[2] *if, for every operation sequence α such that $\alpha \cdot \beta$ and $\alpha \cdot \gamma$ are legal, $\alpha \cdot \beta \cdot \gamma$ and $\alpha \cdot \gamma \cdot \beta$ are legal and equivalent.*

Formally we say that two operations "do not commute" by negating Definition 4. If op_1 and op_2 do not commute, then there exists a sequence of operations α such that $\alpha \cdot op_1$ and $\alpha \cdot op_2$ are legal and (at least) one of the following is true:

1. $\alpha \cdot op_1 \cdot op_2$ is not legal
2. $\alpha \cdot op_2 \cdot op_1$ is not legal
3. $\alpha \cdot op_1 \cdot op_2$ does not look like $\alpha \cdot op_2 \cdot op_1$, which means that there exists an operation sequence γ such that $\alpha \cdot op_1 \cdot op_2 \cdot \gamma$ is legal but $\alpha \cdot op_2 \cdot op_1 \cdot \gamma$ is not
4. $\alpha \cdot op_2 \cdot op_1$ does not look like $\alpha \cdot op_1 \cdot op_2$, which means that there exists an operation sequence γ such that $\alpha \cdot op_2 \cdot op_1 \cdot \gamma$ is legal but $\alpha \cdot op_1 \cdot op_2 \cdot \gamma$ is not

If item 1 or item 2 above is true, then we say that op_1 and op_2 **immediately do not commute**[3]. This implies that OP_1 and OP_2 immediately do not commute. If item 3 above is true, then we say that op_1 and op_2 **eventually do not commute**, and also that OP_1 and OP_2 eventually do not commute. If items 1 and 2 above are true, then we say that op_1 and op_2 are **cyclically dependent**, and also that OP_1 and OP_2 are cyclically dependent.

Appendix A contains examples of several abstract data types with their commutativity properties. We will refer to data types from there throughout this work.

This first theorem gives a condition under which an *individual* (generic) operation of an abstract data type must be slow.

[2] "commute forward" in [Wei93]

[3] If op_1 and op_2 are the same instantiation of the same generic operation, then op_1 **immediately does not commute with itself**, implying that OP_1 immediately does not commute with itself.

Theorem 8.1 *Let T be an abstract data type with a generic operation OP that immediately does not commute with itself. In any sequentially consistent implementation of objects of type T, $|OP| \geq d$.*

Proof The following proof generalizes the proof in [AW94] that a dequeue operation of the queue abstract data type must take at least time d.

Let A be an object of type T. Let processes 1 and 2 access A. Suppose in contradiction that there is a sequentially consistent implementation of A for which $|OP| < d$.

Since OP immediately does not commute with itself, there exist a sequence ρ of operations and an operation instance op such that $\rho \cdot op$ is legal but $\rho \cdot op \cdot op$ is not legal.

We consider A_ρ, the ρ-initialized version of A[4].

By the sequential specification of A_ρ, there is some admissible execution α_1 such that $ops(\alpha_1)$ is $op[A_\rho, 1]$. There is an admissible execution α_2 such that $ops(\alpha_2)$ is $op[A_\rho, 2]$. By assumption, the real times at the end of α_1 and α_2 are less than d. Thus, no process in α_1 or α_2 receives any message. Since no messages are received in α_1 and α_2, replacing p_2's history in α_1 with its history in α_2 results in another admissible execution, α. By assumption, α is sequentially consistent. Thus, there is a τ which is a permutation of $ops(\alpha)$ and is legal for A_ρ. However, all possible permutations of $ops(\alpha)$ are of the form $op \cdot op$, which is not legal for A_ρ. We have a contradiction. ■

The operations in all following corollaries are from the data types in Appendix A.

Corollary 8.1 *The following are true:*

[4]The use of the explicit initialization assumption is necessary in this proof because we must use serialized operation sequences of the form $\rho \cdot \gamma$ to prove a violation of sequential consistency, and the definition of sequential consistency does not require that ρ appear as a prefix of the serialization of the execution. The necessity of the explicit initialization assumption is an open question.

- *In any sequentially consistent implementation of augmented or regular queues (Table A.1),* $|DEQ| \geq d$ *([AW94]).*

- *In any sequentially consistent implementation of augmented or regular stacks (Table A.2),* $|POP| \geq d$ *([AW94]).*

- *In any sequentially consistent implementation of dictionary sets (Table A.3),* $|DEL| \geq d$.

- *In any sequentially consistent implementation of bank account objects (Table A.4),* $|WITHDRAW| \geq d$.

For read/write objects, $write(v)()$ immediately commutes with $write(w)()$, but the two writes eventually do not commute if $v \neq w$. Therefore, the previous theorem does not apply for $|WRITE|$. In fact, [AW94] and [MR92] present algorithms in which writes take less than time d.

This next theorem gives a condition under which a *pair* of distinct (generic) operations from an abstract data type must be slow.

Theorem 8.2 *Let T be an abstract data type, and let OP_1 and OP_2 be distinct generic operations on T which immediately do not commute. In any sequentially consistent implementation of at least two objects of type T, $|OP_1| + |OP_2| \geq d$.*

Proof The following proof generalizes the proof in [AW94] that $|READ| + |WRITE| \geq d$ for read/write objects. This actually proves a theorem that appeared in [LS88]. We do not need the assumption about arbitrary initialization of objects to prove this result.

Since OP_1 and OP_2 immediately do not commute, there is a sequence of operations α and operation instances op_1 and op_2 such that $\alpha \cdot op_1$ and $\alpha \cdot op_2$ are legal, but (without loss of generality) $\alpha \cdot op_1 \cdot op_2$ is not legal.

Let A and B be two objects of type T, and let processes 1 and 2 use A and B. Suppose in contradiction that there is a sequentially consistent implementation of A and B for which $|OP_1| + |OP_2| < d$.

There exists an admissible execution σ_1 with $ops(\sigma_1)$ equal to $\alpha[A,1] \cdot \alpha[B,2] \cdot op_1[A,1] \cdot op_2[B,1]$. Assume that $op_1[A,1]$ starts at real time t, and $op_2[B,1]$ starts immediately after $op_1[A,1]$ finishes. Because we have assumed that the real time after the end of σ_1 is less than $t+d$, no process receives a message during σ_1 after time t about $op_1[A,1]$ or $op_2[B,1]$.

There exists an admissible execution σ_2 with $ops(\sigma_2)$ equal to $\alpha[A,1] \cdot \alpha[B,2] \cdot op_1[B,2] \cdot op_2[A,2]$. Assume that $op_1[B,2]$ starts at real time t, and $op_2[A,2]$ starts immediately after $op_1[B,2]$ finishes. Because we have assumed that the real time after the end of σ_2 is less than $t+d$, no process receives a message during σ_2 after time t about $op_1[A,2]$ or $op_2[B,2]$.

Since no messages are received in σ_1 and σ_2 after time t, replacing process 2's history in σ_1 with its history in σ_2 results in another admissible execution, σ. Then $ops(\sigma)$ consists of the operations $op_1[A,1]$ followed by $op_2[B,1]$, and $op_1[B,2]$ followed by $op_2[A,2]$, where both pairs are preceded by $\alpha[A,1]$ and $\alpha[B,2]$.

By assumption, σ is sequentially consistent. Thus there exists a legal operation sequence τ in which

- the operations in $\alpha[A,1]$ are followed by $op_1[A,1]$, and $op_1[A,1]$ is followed by $op_2[B,1]$
- the operations in $\alpha[B,2]$ are followed by $op_1[B,2]$, and $op_1[B,2]$ is followed by $op_2[A,2]$

Since $\alpha \cdot op_1 \cdot op_2$ is not legal, τ must have $op_1[A,1]$ follow $op_2[A,2]$. But that causes $op_2[B,1]$ to follow $op_1[B,2]$, which is not legal.

■

Corollary 8.2 *The following are true:*

- *In any sequentially consistent implementation of read/write objects,* $|READ| + |WRITE| \geq d$ *([LS88, AW94]).*

- *In any sequentially consistent implementation of augmented or regular queues (Table A.1),* $|ENQ| + |PEEK| \geq d$, $|PEEK| + |DEQ| \geq d$, *and* $|ENQ| + |DEQ| \geq d$.

- *In any sequentially consistent implementation of augmented or regular stacks (Table A.2),* $|PUSH| + |PEEK| \geq d$, $|PEEK| + |POP| \geq d$, *and* $|PUSH| + |POP| \geq d$.

- *In any sequentially consistent implementation of dictionary sets (Table A.3),* $|INS| + |DEL| \geq d$, $|INS| + |SEARCH| \geq d$, *and* $|SEARCH| + |DEL| \geq d$.

- *In any sequentially consistent implementation of bank account objects, (Table A.4),* $|DEPOSIT| + |WITHDRAW| \geq d$, $|DEPOSIT| + |BALANCE| \geq d$, *and* $|BALANCE| + |WITHDRAW| \geq d$.

The next theorem shows that cyclic dependences cause a pair of operations to be slow, even slower than the previous lower bound indicates.

Theorem 8.3 *Let T be an abstract data type with generic operations OP_1 and OP_2 that are cyclically dependent. In any sequentially consistent implementation of objects of type T, $|OP_1| + |OP_2| \geq 2d$.*

Proof Since OP_1 and OP_2 are cyclically dependent, they immediately do not commute. Thus, $|OP_1| + |OP_2| \geq d$ by Theorem 8.2.

Since OP_1 and OP_2 are cyclically dependent, there is a sequence of operations ρ and operation instances op_1 and op_2 such that $\rho \cdot op_1$ and $\rho \cdot op_2$ are legal, but $\rho \cdot op_1 \cdot op_2$ and $\rho \cdot op_2 \cdot op_1$ are not legal.

Let A be an object of type T. Let processes 1 and 2 access A. We consider A_ρ, the ρ-initialized version of A.

Assume in contradiction that there exists a sequentially consistent implementation of A for which $|OP_1| + |OP_2| < 2d$. We can assume without loss of generality that $|OP_1| \geq |OP_2|$.

By the sequential specification for A_ρ, there is some admissible execution α_1 such that $ops(\alpha_1)$ is $op_1[A_\rho, 1]$. Assume that the op_1 operation starts at time 0. Then the real time at the end of α_1 is at most $|OP_1|$.

By the sequential specification for A_ρ, there is some admissible execution α_2 such that $ops(\alpha_2)$ is $op_2[A_\rho, 2]$. Assume that the op_2 operation starts at time $(|OP_1| - |OP_2|)/2$. Then the real time after the end of α_2 is at most $(|OP_1| - |OP_2|)/2 + |OP_2|$, which is less than d. Any message sent by process 2 would not be delivered until at least time $(|OP_1| - |OP_2|)/2 + d$, which is more than $|OP_1|$.

Since no messages are received in α_1 and α_2 before time d, replacing process 1's history in α_2 with its history in α_1 results in another admissible execution, α. By assumption, α is sequentially consistent. Thus, there is a permutation of $ops(\alpha)$ which is legal for A_ρ. However, because of cyclic dependency, neither permutation of $ops(\alpha)$ is legal for A_ρ. We have a contradiction.

■

Dequeue and enqueue are not cyclically dependent because an enqueue is always legal after a dequeue. Read and write are not cyclically dependent because a write is always legal after a read. Thus, we cannot strengthen the lower bound for $|DEQ| + |ENQ|$ and $|READ| + |WRITE|$. In fact, [AW94] showed that the lower bounds were tight, displaying algorithms with worst-case response times for the pairs of operations matching the lower bounds of d.

None of the abstract data types discussed so far have any operations which have cyclic dependences with other operations.

We now give an example of a type, TWOCYCLE, with cyclically dependent operations. The object of the type is a two-element array. There are two operations, $R1W2$ and $R2W1$. $r1w2(v)(w)$ writes v to the second element of the array and returns the value of the first element of the array in w. $r2w1(v)(w)$ writes v to the first element of the array and returns the value of the second element of the array in w. $R1W2$ commutes with itself, and $R2W1$ commutes with itself. However, $R1W2$ and $R2W1$ are cyclically dependent.

We give another example of a type, TWOFIVE, with cyclically dependent operations. The object of the type is a variable which holds a real number. There are

two operations, $ADD2EVEN$ and $ADD5DIV5$. $add2even()(v)$ causes the object's value to be incremented by 2 and returns the parity of the object's previous value in v (0 for odd, 1 for even). $add5div5()(v)$ cause the object's value to be incremented by 5 and returns a value denoting the divisibility of the object's previous value by 5 (0 if not divisible by 5, 1 otherwise). $ADD2EVEN$ commutes with itself, and $ADD5DIV5$ commutes with itself. However, $ADD2EVEN$ and $ADD5DIV5$ are cyclically dependent.

Corollary 8.3 *The following are true:*

- *In any sequentially consistent implementation of objects of type TWOCYCLE,* $|R1W2| + |R2W1| \geq 2d$.
- *In any sequentially consistent implementation of objects of type TWOFIVE,* $|ADD2EVEN| + |ADD5DIV5| \geq 2d$.

We now define a condition for a trio of operations which could cause either a pair of operations or an individual operation to be slow.

A generic operation OP is **noninterleavable with respect to** OP_1 **preceding** OP_2 if there exist operation sequence ρ and operation instances op, op_1, and op_2 such that $\rho \cdot op$ and $\rho \cdot op_1 \cdot op_2$ are legal, but none of $\rho \cdot op \cdot op_1 \cdot op_2$, $\rho \cdot op_1 \cdot op \cdot op_2$, and $\rho \cdot op_1 \cdot op_2 \cdot op$ is legal.

Theorem 8.4 *Let T be an abstract data type, and let OP_1, OP_2, and OP_3 be generic operations of T such that OP_3 is noninterleavable with respect to OP_1 preceding OP_2. Then in any sequentially consistent implementation of objects of type T, $|OP_1| + |OP_2| \geq d$ or $|OP_3| \geq d$.*

Proof Since OP_3 is noninterleavable with respect to OP_1 and OP_2, there exists an operation sequence ρ and operation instances op_1, op_2, and op_3 such that $\rho \cdot op_3$ and $\rho \cdot op_1 \cdot op_2$ are legal, but none of $\rho \cdot op_3 \cdot op_1 \cdot op_2$, $\rho \cdot op_1 \cdot op_3 \cdot op_2$, and $\rho \cdot op_1 \cdot op_2 \cdot op_3$ is legal.

Let A be an object of type T. Let processes 1 and 2 access A. We consider A_ρ, the ρ-initialized version of A.

Assume in contradiction that there exists a sequentially consistent implementation of A for which $|OP_1| + |OP_2| < d$ and $|OP_3| < d$.

By the sequential specification for A_ρ, there is some admissible execution α_1 such that $ops(\alpha_1)$ is $op_1[A_\rho, 1] \cdot op_2[A_\rho, 1]$. Assume that the op_1 operation starts at time 0 and that the op_2 operation starts immediately after the op_1 operation finishes. Because the real time after the end of α_1 is less than d, no process receives a message during α_1.

By the sequential specification for A_ρ, there is some admissible execution α_2 such that $ops(\alpha_1)$ is $op_3[A_\rho, 2]$. Assume that the op_3 operation starts at time 0. Because the real time after the end of α_2 is less than d, no process receives a message during α_2.

Since no messages are received in α_1 and α_2, replacing process 1's history in α_2 with its history in α_1 results in another admissible execution, α. By assumption, α is sequentially consistent. Thus, there is a permutation of $ops(\alpha)$, τ, which preserves the order of process 1's operations in α and is legal for A_ρ. However, because of noninterleavability, none of the three permutations of $ops(\alpha)$ which satisfies the order of process 1's operations is legal for A_ρ. We have a contradiction. ■

Let TWOARRAY be an abstract data type where the objects are two-element arrays and the operations are $R1W2$ (as in TWOCYCLE), $W1$ (which writes to the first element of the array on which it is invoked), and $R2$ (which reads and returns the value of the second element of the array on which it is invoked). $R1W2$ is noninterleavable with respect to $W1$ preceding $R2$.

Corollary 8.4 *In any sequentially consistent implementation of objects of type TWOARRAY, $|W1| + |R2| \geq d$ or $|R1W2| \geq d$.*

8.2 Upper Bounds Where a Single Operation or Class of Operations is Optimized

We have shown several lower bounds on the worst-case time required for operations in linearizable implementations of objects from general abstract data types. We

used various algebraic properties of the operations to prove these lower bounds. In particular, we proved that the worst-case completion time of an operation is at least d if it immediately does not commute with itself. Given an operation that immediately commutes with itself, can we optimize its completion time in a linearizable implementation of its abstract data type? In other words, can we implement it so that it only performs local computation, for which the time is assumed to be negligible compared to the message delay in the communication network? We now attempt to answer this question. Optimizing an operation's completion time is useful if the operation is used frequently in applications. We demonstrate three algebraic properties such that an operation or group of operations can be optimized if it satisfies one of these properties. Optimizing just a single operation from an abstract data type has proved to be a nontrivial task. Although we have not quite developed a tight characterization about exactly when a single operation can be optimized, we have shown that if an operation is *self-oblivious*, then that operation can be optimized. The self-oblivious property is a common property for operations. From our lower bound results, we know that any operation which immediately does not commute with itself cannot be optimized. We also know that if an operation is noninterleavable with respect to a pair of operations, then some slowdown must occur (either the individual operation or the pair). The absence of the self-oblivious property is related to both of these properties. If an operation immediately does not commute with itself, then it is not self-oblivious. If an operation is not self-oblivious, then it may be noninterleavable with respect to another operation and itself.

We now define some more properties of generic operations. Let α and β be arbitrary operation sequences. If the legality of $\alpha \cdot aop \cdot \beta$ implies the legality of $\alpha \cdot \beta$ for any instance aop of generic operation AOP and the legality of $\alpha \cdot \beta$ implies the legality of $\alpha \cdot aop^* \cdot \beta$ for some instance aop of generic operation AOP (where op^* denotes 0 or more copies of op), then AOP is an **accessor**. Informally speaking, an accessor does not change the state of an object. MOP is a **modifier** if there exist operation sequences α and β such that $\alpha \cdot mop \cdot \beta$ is legal but $\alpha \cdot \beta$ is not legal for some instance mop of MOP. Informally speaking, a modifier changes the state of an object, and this change can be detected.

For any abstract data type, we can always implement accessor operations such that they only perform local computation.

Theorem 8.5 *Let T be an abstract data type. If T has generic accessor operations $AOP_1, \ldots, AOP_n$, then there exists a linearizable implementation of objects of type T where $|AOP_1| = \ldots = |AOP_n| = 0$ and $|MOP_1| = \ldots = |MOP_m| = d$ for all other generic operations $MOP_1, \ldots, MOP_m$.*

Proof We exhibit an implementation where $|AOP_1| = \ldots = |AOP_n| = 0$ and $|MOP_1| = \ldots = |MOP_m| = d$. Each process keeps a copy of all objects in its local memory. When an aop_i is invoked at process p, p performs the operation locally and returns the result from the operation. When a mop_j operation on object X is invoked at process p, p sends a message $DoMop_j(X)$ with the argument list for the operation to all processes (including itself), waits d time, and returns the result of performing the operation. When a process receives any form of $DoMop$ message, it performs the operation on X in its local memory. If the message was sent by that process, it saves the result so that the process can return it. We can break ties in the following way. $DoMop_k$ is handled before $DoMop_l$ if $k < l$, and we use process identifiers to break any remaining ties.

We now prove that this algorithm is correct. The proof is very similar to a proof in [AW94] for read/write objects.

Let ρ be an admissible execution. We systematically construct the desired τ. Each operation in ρ *occurs* at the time of its response. Let τ be the sequence of operations in ρ ordered by the times of occurrence, breaking ties by placing mop operations before aop operations, aop_k before aop_l if $k < l$, mop_k before mop_l if $k < l$, and using process identifiers to break any remaining ties. By construction, $\rho|p = \tau|p$ for all p, and τ preserves the relative ordering of non-overlapping operations.

We now must show that τ is legal, or that for each object X, $\tau|X$ is in the sequential specification of T.

Consider an accessor operation. It returns based on its local state. Its local state reflects all changes made by modifiers occurring up to the time of the accessor in ρ. Thus the accessor operation returns a legal value list in τ.

Consider a modifier operation that returns a value list. It returns based on its local state at its response time in ρ. Its local state reflects all changes made by modifiers occurring up to the time of the modifier. Let the sequence of modifiers occurring up to the time of the modifier in ρ be α. The modifier is legal after α, which is a subsequence of τ. Any accessors interleaved with α in τ will not affect the legality of the modifier. Thus the modifier operation returns a legal value list in τ. ■

We now define an algebraic property of modifier operations which more finely classifies them. Let α be a sequence of operations. If the legality of α implies the legality of $\alpha \cdot mop$ for any instance mop of generic modifier operation MOP, then MOP is a **pure modifier**. Informally speaking, pure modifiers are modifiers whose return value lists do not depend on the states of the objects on which they are invoked.

For any abstract data type, we can always implement pure modifier operations so that they return immediately.

Theorem 8.6 *Let T be an abstract data type. If T has generic pure modifier operations $MOP_1, \ldots, MOP_n$, then there exists a linearizable implementation of objects of type T where $|MOP_1| = \ldots = |MOP_n| = 0$ and $|OP_1| = \ldots = |OP_m| = d$ for all other generic operations $OP_1, \ldots, OP_m$.*

Proof We exhibit an implementation where $|MOP_1| = \ldots = |MOP_n| = 0$ and all other operations $(OP_1, \ldots, OP_m)$ take time d. Each process keeps a copy of all objects in its local memory. When a mop_i is invoked at process p, p sends a message $DoMop_i(X)$ with the argument list for the operation to all processes (including itself) and returns immediately. When a process receives any form of $DoMop$ message, it performs the operation on X in its local memory. When an op_j on object X is invoked at process p, p sends a message $DoOp_j(X)$ with the argument list for the operation to all processes (including itself), waits d time, and returns the result of performing the operation. When a process receives any form of Op message, it performs the operation on X in its local memory. If the message was sent by that process, it saves the result so that the process can return it. We

can break ties in the following way. $DoMop$ messages are handled before $DoOp$ messages, $DoMop_k$ is handled before $DoMop_l$ if $k < l$, $DoOp_k$ is handled before $DoOp_l$ if $k < l$, and we use process identifiers to break any remaining ties.

We now prove that this algorithm is correct. The proof is very similar to a proof in [AW94] for read/write objects.

Let ρ be an admissible execution. We systematically construct the desired τ. Each operation in ρ *occurs* time d after the time of its call. Let τ be the sequence of operations in ρ ordered by the times of occurrence, breaking ties by placing *mop* operations before *op* operations, mop_k before mop_l if $k < l$, op_k before op_l if $k < l$, and using process identifiers to break any remaining ties. By construction, $\rho|p = \tau|p$ for all p, and τ preserves the relative ordering of non-overlapping operations.

We now must show that τ is legal, or that for each object X, $\tau|X$ is in the sequential specification of T.

All *mop* operations are legal because they are pure modifiers.

Consider an op_k that returns a value list. It returns based on its local state at its response time in ρ. Its local state reflects all changes made by operations occurring up to the time of the op_k in ρ. Thus the op_k returns a legal value list in τ. ■

We have just shown how we can optimize pure accessors and pure modifiers. A natural question to ask now is the following: Can we optimize a self-commuting operation which both accesses and modifies the states of the objects on which it is invoked?

We must be very careful because two fast operation instances will not know about each other if they are executed less than time d apart. Given a generic operation FOP, in order to optimize $|FOP|$, it is sufficient for FOP to be **self-oblivious**. Intuitively, this means that a fop operation instance will not indirectly affect another fop operation instance. We now give a formal definition.

Definition 5 *Let $\alpha_1, \alpha_2, \ldots$ be sequences of operations. FOP is* **self-oblivious** *if whenever $\alpha_1 \cdot fop^1$, $\alpha_1 \cdot \alpha_2 \cdot fop^2$, ..., $\alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_n \cdot fop^n$, ... are legal, there*

exists an instantiation of return values for the operations in α_i for each $i \geq 1$, creating new sequences of operations α'_i for each $i \geq 1$, such that $\alpha'_1 \cdot fop^1 \cdot \alpha'_2 \cdot fop^2 \cdot \ldots \cdot \alpha'_n \cdot fop^n \ldots$ is legal.

We can observe that an operation which immediately does not commute with itself is not self-oblivious because the α's can be empty sequences.

Given an abstract data type with a self-oblivious operation, can we optimize the worst-case response time for that operation in a linearizable implementation of the abstract data type? We now answer that question in the affirmative.

Theorem 8.7 *Let T be an abstract data type with a self-oblivious generic operation $SELFOP$. There exists a linearizable implementation of objects of type T where $|SELFOP| = 0$ and $|OP| = 2d$ for all other generic operations OP.*

Proof We instantiate the algorithm in Figures 8.1 and 8.2 to yield our implementation of objects of type T. We note that each assignment statement is executed locally.[5] Each process keeps both an actual copy and a scratch copy of each object. Each process also maintains an ordered set of message slots, where each slot is indexed by a time. When a $selfop$ operation is invoked, the invoking process sends a message about the operation to all other processes, determines the return value list (by possibly updating its scratch copy of the object based on messages it has received), and returns. When an instance of another generic operation is invoked, the invoking process sends a message about the operation to all other processes. If a message about a self-oblivious operation is received at time t, then it is placed in slot $t - d$. If any other message is received at time t, then it is placed in slot t. If a received message is not about a $selfop$ operation, then a timer for d later is set. When a timer goes off, all messages in slots indexed by times up to d before the current time are handled, updating actual copies of objects as necessary. At this time, if the process has a pending operation for which its message has been handled, the process completes its pending operation.

We must now show that the implementation guarantees linearizable executions. Let ρ be an admissible execution. To form τ, we place all operations in order

[5] In all subsequent descriptions of algorithms, assignment statements are also executed locally.

according to their message slots, and according to their positions in the message slots if their message slots contain multiple messages.

In τ, the relative ordering of nonoverlapping operations is preserved because of the constant message delay d.

We must now show that τ is legal. Choose an object X and consider $\tau|X = op_1 op_2 \ldots$. Consider op_i. If op_i is a $selfop$ operation, then op_i's return value is based on the sequence of operations that have been executed on $scratch_X$ at op_i's process by the time op_i was invoked in ρ. This sequence is a subsequence of τ. Since op_i is self-oblivious, op_i's return value is legal in τ. If op_i is not a $selfop$ operation, then its return value is based on the sequence of operations that have executed at op_i's process in ρ by the time op_i returned in ρ. This sequence is a prefix of τ. Thus, op_i is legal in τ.

■

What kinds of operations are self-oblivious? Any accessor operation is self-oblivious. Any pure modifier operation is self-oblivious because it is always legal after a legal sequence of operations. However, determining whether an arbitrary operation is self-oblivious requires looking at the semantics for all operations of its abstract data type. An operation that immediately commutes with itself is self-oblivious if all other operations of its abstract data type do not perform conditional updates (whether and how to perform updates are based on object state information).

Let us give some specific examples of self-oblivious operations. For read/write objects, reads and writes are self-oblivious. For queues and stacks, enqueues and pushes are self-oblivious. For TWOARRAY objects, $R1W2$ and $R2W1$ are self-oblivious.

In our implementation optimizing a self-oblivious operation, all non-optimized operations have a worst-case response time of $2d$. Given an abstract data type with such an operation to be optimized, how slow must the other operations be? Since a self-commuting operation may have cyclic dependences with other operations, by the result of Theorem 8.3, these other operations must have a worst-case time

complexity which is at least $2d$. If this self-commuting operation immediately does not commute with some other operation, then the other operation must have a worst-case time complexity which is at least d.

8.3 Lower Bounds for All Operations of a Type

In Section 8.1, we determined lower bounds for single operations and pairs of operations from abstract data types. We use the results from that section and the structure of the commutativity graphs to deduce lower bounds on the worst-case time complexity for *all* operations of abstract data types in sequentially consistent implementations.

An alternative way to represent the commutativity properties of an abstract data type T is to use a **commutativity graph** $CG(T)$, where the nodes are the generic operations. There exists an edge between two nodes if their corresponding operations immediately do not commute. There exists a loop at a node if the corresponding operation immediately does not commute with itself. We let $NSC(T)$ be the subset of nodes in $CG(T)$ such that each node's corresponding operation immediately does not commute with itself. We let $RCG(T)$ (the reduced commutativity graph for T) be the subgraph of $CG(T)$ formed by deleting all nodes in $NSC(T)$ and their incident edges. We let $Maxdom(RCG(T))$ be a subgraph of $RCG(T)$ formed by a maximum independent edge dominating set of $RCG(T)$.[6]

We now give a lower bound on the time required for all operations of an abstract data type with a *clique* in its commutativity graph.

Theorem 8.8 *Let T be an abstract data type with operations $OP_1, OP_2, \ldots, OP_n$ such that for all $i \in \{1, \ldots, n\}$, OP_i immediately does not commute with OP_j if $i \neq j$. In any sequentially consistent implementation of T, $\sum_{i=1}^{n} |OP_i| \geq sd + (n - s)d/2$, where s is the number of operations which immediately do not commute with themselves.*

[6] A maximum independent edge dominating set of a graph is a largest subset of edges of the graph such that distinct edges in the subset do not have nodes in common and all other edges in the graph have a node in common with one of the edges in the set.

Proof Let $OP_{i_1}, \ldots, OP_{i_s}$ be the operations which immediately do not commute with themselves. By Theorem 8.1, $|OP_{i_k}| \geq d$ for all k in $\{1, \ldots, s\}$. Thus, $\sum_{k=1}^{s} |OP_{i_k}| \geq sd$. Let $OP_{j_1}, \ldots, OP_{j_{n-s}}$ be the remaining operations. We can assume without loss of generality that $t = n - s > 2$ because Theorem 8.2 handles the $t = 2$ case. From the result of Theorem 8.2, we get a system of $t(t-1)/2$ equations of the form $|OP_{j_a}| + |OP_{j_b}| \geq d$, where $a \neq b$. We want to minimize $\sum_{k=1}^{t} |OP_{j_k}|$ given the above constraints and the constraints $|OP_{j_k}| \geq 0$ for all $k \in \{1, \ldots, t\}$. Set $|OP_{j_a}| + |OP_{j_b}| = d$. Without loss of generality we consider

- $|OP_{j_1}| + |OP_{j_2}| = d$
- $|OP_{j_1}| + |OP_{j_3}| = d$
- $|OP_{j_2}| + |OP_{j_3}| = d$

We can subtract the third equation from the second equation to yield the equation $|OP_{j_1}| - |OP_{j_2}| = 0$. Adding the new equation to the first yields $2|OP_{j_1}| = d$. We now get that $|OP_{j_1}| = d/2$. It easily follows that $|OP_{j_k}| = d/2$ for all $k \in \{1, \ldots, t\}$. Thus, $\sum_{k=1}^{t} |OP_{j_k}| \geq (n-s)d/2$. Adding these two inequalities yields the desired result. ■

Corollary 8.5 *The following are true:*

- *In any sequentially consistent implementation of reference-count sets,* $|INS| + |UP| + |FIND| \geq 3d/2$.
- *In any sequentially consistent implementation of a reference-count set with a delete operation,* $|DEL| + |INS| + |UP| + |FIND| \geq d + 3d/2 = 5d/2$.
- *In any sequentially consistent implementation of a bounded double-ended peek queue (where the peek operation returns the contents at each end of the queue),* $|BACKDEQ| + |FRONTDEQ| + |BACKENQ| + |FRONTENQ| + |PEEK| \geq 2d + 3d/2 = 7d/2$.

The previous theorem gave lower bounds on the costs of implementing abstract data types with cliques in their commutativity graphs. This next theorem gives

lower bounds on the costs of implementing abstract data types with more general commutativity graphs.

Theorem 8.9 *Let T be an abstract data type with operations $OP_1, OP_2, \ldots, OP_n$ and commutativity graph $CG(T)$. In any sequentially consistent implementation of T, $\sum_{i=1}^{n} |OP_i| \geq (|NSC(T)| + |Maxdom(RCG(T))|)d$.*

Proof If OP_i immediately does not commute with itself, then $|OP_i| \geq d$ by Theorem 8.1. Thus, $\sum_{OP_i \in NSC(T)} |OP_i| \geq |NSC(T)|d$. Let (OP_i, OP_j) be an edge in $Maxdom(RCG(T))$. By Theorem 8.2, $|OP_i| + |OP_j| \geq d$. By adding together these inequalities, we obtain the desired result. ■

8.4 Types Having a Tight Lower Bound for All Operations

We now exhibit some abstract data types for which there are implementations in which the total time for all operations matches the lower bounds proved in the previous subsection. Minimizing the total time required for all operations may help when the frequencies of invoking each operation are approximately equal.

A pure modify-read (PMR) object is a variable X that can be read or modified by a pure modifier operation. This is a generalization of the pseudo read-modify-write (PRMW) object [AG91] (a variable that can be read, written, or modified by a pure modifier operation that is a commutative arithmetic operation) because the pure modifier operation may be such that it eventually does not commute with itself.

We now show that there exists an implementation of PMR objects with the total worst-case response time for all operations matching the lower bound on the total worst-case response time for all operations.

Theorem 8.10 *For any set of PMR objects with operations $READ$ and $MOP_1, \ldots, MOP_n$ (pure modifier operations), there exists a linearizable implementation of the set with $|READ| = d$ and $|MOP_1| = \ldots = |MOP_n| = 0$.*

Proof Since all non-read operations are pure modifiers, we can instantiate the implementation described in Theorem 8.6 to yield an implementation achieving the desired time bounds. ■

Corollary 8.6 *There exists a linearizable implementation of a set of Increment-Half objects (Table A.6), where* $|READ| = d$, $|INC| = 0$, *and* $|HALF| = 0$.

We now show why the upper bound for all operations matches the lower bound for all operations in linearizable implementations of PMR objects. By Theorem 8.2, $|READ| + |MOP_i| \geq d$ for each i in $\{1, \ldots, n\}$. Thus, $|READ| + |MOP_1| + \ldots + |MOP_n| \geq d$, matching the upper bound from Theorem 8.10. All pairs of distinct non-read operations may be such that they eventually do not commute, but they immediately do commute; thus the lower bound from Theorem 8.2 does not apply for them.

PMR objects have sparse commutativity graphs. All nodes have degree 1, with the exception of the READ node. Can we optimize the total worst-case time complexity for abstract data types with dense commutativity graphs? We give a specific data type for which the answer is affirmative.

The reference-count set abstract data type (Table A.5) has a complete commutativity graph, which has maximum density. Corollary 8.5 gave a lower bound of $3d/2$ on the total time complexity for sequentially consistent implementations of reference-count sets. Can we get a matching upper bound? We now answer this question in the affirmative.

Theorem 8.11 *There exists a linearizable implementation of reference-count sets in which* $|INS| = d/2$, $|UP| = d/2$, *and* $|FIND| = d/2$.

Proof We now describe an algorithm which achieves the above bounds. Each process keeps a copy of every set in its local memory. Figures 8.3 and 8.4 contain a description of the algorithm. When a process invokes an operation, it sends a message about the operation if it is a modifier, and it waits $d/2$ time before returning based on its local state. If a process receives several messages at the same time, it

handles insert messages before update messages, breaking any other ties by using process identifiers. When a process handles a message, it updates its local state accordingly.

We must now show that this algorithm produces linearizable executions.

Let ρ be an admissible execution. We show how to produce τ. We serialize normal updates (updates not returning $\perp$) and inserts to occur $d/2$ after their response times, and finds and updates returning $\perp$ (abnormal) to occur at their response times, breaking ties by ordering inserts before updates before finds and using process identifiers to break the remaining ties. By construction, $\tau|p = \rho|p$ for all processes p, and τ preserves the relative order of non-overlapping operations.

We now show that τ is legal. We have the following 6 cases to check.

1. An abnormal update cannot be placed after a normal update for the same object. Suppose in contradiction that an abnormal update is placed after a normal update for the same object. Let t_1 be the response time in ρ for the abnormal update, and let t_2 be the response time in ρ for the latest such normal update placed before the abnormal update. By the definition of τ, the normal update's serialization time is $t_2 + d/2$, and the abnormal update's serialization time is t_1. Since the abnormal update is placed later, $t_1 \geq t_2 + d/2$. However, $t_2 = t_{call} + d/2$, where t_{call} is the time of the call for the normal update. Thus $t_1 \geq t_{call} + d$, which means that the normal update should have happened at the process which invoked the abnormal update. We have a contradiction.

2. An abnormal update cannot be placed after an insert for the same object. The argument is similar to the argument for the previous case.

3. An abnormal find cannot be placed after an insert for the same object. The argument is similar to the argument for Case 1.

4. A normal find cannot be placed before the first insert for the same object. Suppose in contradiction that a normal find is placed before the first insert for the same object. Let t_1 be the response time for the first insert. Let t_2 be the response time for the find. By the definition of τ, the insert's serialization

time is $t_1 + d/2$, and the find's serialization time is t_2. Since the first insert is placed later, $t_2 < t_1 + d/2 = (t_{call} + d/2) + d/2 = t_{call} + d$. Thus the process that invoked the find could not have known that an insert was performed, and the find must have been abnormal. We have a contradiction.

5. A normal update cannot be placed before the first insert for the same object. Suppose in contradiction that a normal update is placed before the first insert for the same object. Let t_1 be the response time in ρ for the insert, and let t_2 be the response time in ρ for the update. By the definition of τ, the insert's serialization time is $t_1 + d/2$, and the update's serialization time is $t_2 + d/2$. Since the update is placed before the insert, $t_2 + d/2 < t_1 + d/2 = (t_{call} + d/2) + d/2 = t_{call} + d$, where t_{call} is the time of the call for the insert. Thus $t_2 < t_{call} + d/2 < t_{call} + d$. The process invoking the update had not received any information about the insert. Thus the update is not normal, a contradiction.

6. The value returned by a normal find must be 1 more than the number of normal updates for the same object that are placed before the find. By Case 4, the value returned by a normal find is at least 1. Suppose that the value returned by a normal find is greater than 1 plus the number of normal updates for the same object that are placed before the find. Then there must be an update for the same object that is placed after the find. Let t_1 be the response time in ρ for the find, and let t_2 be the response time in ρ for the update placed after the find. By the definition of τ, the find's serialization time is t_1, and the update's serialization time is $t_2 + d/2$. Since the update is placed after the find, $t_2 + d/2 > t_1$. But $t_2 + d/2 = (t_{call} + d/2) + d/2 = t_{call} + d$, where t_{call} is the time of the call for the update. Thus the process that invoked the find could not have known that another update was performed and thus could have not returned a greater value. Now suppose that the value returned by a normal find is less than 1 plus the number of normal updates for the same object that are placed before the find. Let t_i be the response time in ρ for the i^{th} normal update placed before the find. By the definition of τ, the update's serialization time is $t_i + d/2$. Since the update is placed before the find, $t_i + d/2 \leq t_1$. But $t_i + d/2 = (t_{call} + d/2) + d/2 = t_{call} + d$. Thus the

process invoking the find must have known about each update. We have a contradiction.

■

$fastop(args)[X,p]$:

$scratch_X := X$

Handle all unhandled messages in slots up to (current time) about operations on X in message slot order, updating $scratch_X$ as necessary

Determine return value based on $scratch_X$

Send message handle-fastop$(X, args, p)$ to all processes

$fastopret(returnvalue)[X,p]$

$otherop(args)[X,p]$:

send message handle-otherop$(X, args, p)$ to all processes

receive a message:

if message is a handle-fastop message then

Insert in slot (current time - d) with tiebreaking after other types of messages and then by process identification

else handle-otherop message

Insert in slot (current time)

with a fixed tiebreaking order among different types of messages and then by process identification

timerset(p, d)

end if

Figure 8.1: Algorithm for Optimizing One Operation - Code for Process p

```
when timer goes off:
    for each unhandled message in slots up to (current time - d), considered in message
    slot order, do:
        Use it to perform an update on the actual copy of its object
        if it is not a handle-fastop message then
        /* it is of the form handle-otherop(X, args, q) */
            if p = q then
                Determine return value based on X
                otheropret(returnvalue)[X, p]
            end if
        end if
        Mark message as handled
```

Figure 8.2: Algorithm for Optimizing One Operation - Code for Process p (continued)

```
Ins(x)[S,p]:
        timerset(p, d/2)
        broadcast(DoInsert(x,S))
        when timer expires:
            Ack(Ok)[S,p]

DoInsert_p(x,S): /* executed when message is received */
        if an element with key x is not already in S then
                insert element (x,1) into S
        end if

Up(x)[S,p]:
        timerset(p, d/2)
        if an element with key x is already in S then
            broadcast(DoUpdate(x,S))
        end if
        when timer expires:
            if an element with key x is not already in S then
                Ack(⊥)[S,p]
            else
                Ack(Ok)[S,p]
            end if

DoUpdate_p(x,S): /* executed when message is received */
        Find element (x,v) in S.
        (x,v) := (x,v+1)
```

Figure 8.3: Algorithm for Reference-Count Set - Code for Process p

$Find(x)[S,p]$:

timerset$(p, d/2)$

when timer expires:

if an element with key x is in S then

$Ret(v)[S,p]$, where v is the value of the element with key x

else

$Ret(\perp)[S,p]$

end if

Figure 8.4: Algorithm for Reference-Count Set - Code for Process p (continued)