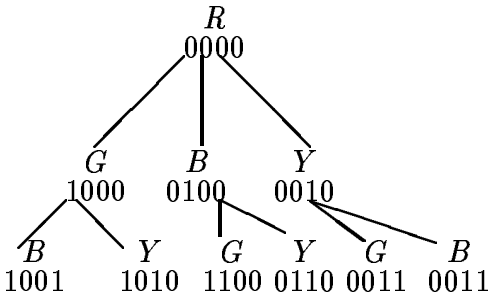

Figure 4.9: Case 6.1

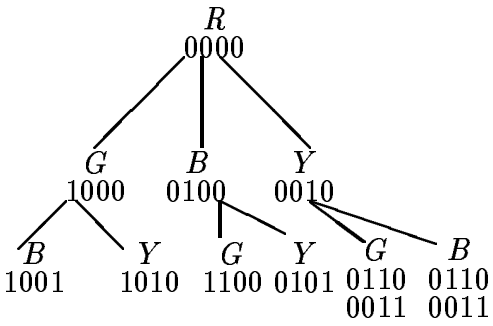

Figure 4.10: Case 6.2

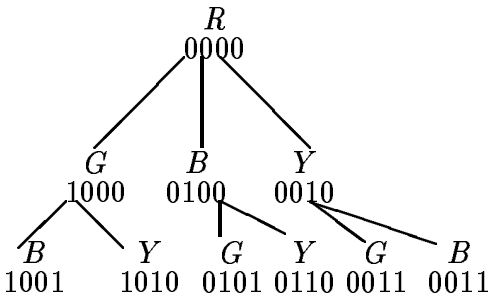

Figure 4.11: Case 6.3

Lemma 4.3 *Consider any $A \in SYM(k)$ with $R_A(k) = m$. Suppose there exists a reachable configuration C and a value $w \neq f_A(C)$ such that C has p neighbors D with $f_A(D) = w$. Then there exists a one-write algorithm $A' \in SYM(k-1)$ with $R_{A'}(k-1) \leq m - p$.*

Proof We show how to construct A' given A. A' will implement a logical register with value set $V - \{w\}$, where V is the value set of the logical register implemented by A, and initial value $v_0 \in V - \{w\}$.

For each $i \in \{1, \ldots, p\}$, let C_i be the neighbor of C that differs from C in bit b_i, where $f_A(C_i) = w$. Consider the set S of all configurations L reachable from C by a path of algorithm edges in which no configuration X with $f_A(X) = w$ appears in the path. Let Z be the subgraph of H_A in which the node set is S and the edge set is the set of all edges in $S \times S$ that are algorithm edges in H_A. No edge in Z is labeled with any bit in $\{b_1, b_2, \ldots, b_p\}$ because otherwise some C_i is constructible from S, which is connected, and $f_A(C_i) = w$ is not in $f_A(S)$, contradicting Lemma 4.1.

Algorithm A' will use $m - p$ binary regular registers. We now define the initial configuration for A'. Assume without loss of generality that b_1 through b_p are the last p bits and they are all 0 in C. Thus, b_1 through b_p are all 0 in every configuration in S. Given $D \in S$, define $\pi(D)$ to be the prefix of D consisting of all but the last p bits. (These will be the reachable configurations of A'.) If $f_A(C) = v_0$, let $D_0 = C$. Otherwise, let D_0 be the neighbor of C in Z such that $f_A(D_0) = v_0$. Clearly D_0 exists. We define the initial configuration of A' to be $\pi(D_0)$.

We now describe the reader's protocol in algorithm A'. The reader's protocol in algorithm A' is the same as the reader's protocol in algorithm A, except that the reader in A' has local bits $c_1, \ldots, c_p$ corresponding to shared bits $b_1, \ldots, b_p$ in A. The value of bit c_i is 0 for each $i \in \{1, \ldots, p\}$ at all times. Whenever reader j in A reads shared bit b_i, the reader in A' reads local bit c_i using action localread(j, c_i).

We now describe the writer's protocol in algorithm A'. If the current configuration of the physical registers (well-defined because readers do not write) is $\pi(E)$ for some $E \in S$ and if WRITE(x), for x not the current value of the logical register, is the next operation, then the writer changes bit b, where b labels the algorithm edge (E, D) in Z and $f_A(D) = x$. An easy induction shows that in every state of

every execution of A' the physical registers always form a configuration $\overline{E}$ such that $\overline{E} = \pi(E)$ for some $E \in S$.

Now we must show that algorithm A' implements a $(k-1)$-ary regular register. Algorithm A' clearly holds $(k-1)$ values and satisfies the wait-free property. We now show that the regular property holds. Consider any execution e' of algorithm A'. We build a corresponding execution e of algorithm A as follows. We construct a sequence of actions of A by starting with a sequence of logical WRITEs to ensure that the configuration of the physical registers is D_0. We then consider each action in the execution of A' in turn. If the action is not a read of a local bit c_i by reader j, then the action is placed as is in the sequence. If the action is a read of a local bit c_i by reader j, then the actions $\text{read}_{b_i}(j)$ and $\text{return}_{b_i}(j,0)$ are placed in order in the sequence. By induction, there exists an execution e of A with the sequence of actions just constructed. By the assumption about A, e satisfies the regular property. Suppose a READ by reader j in execution e' of algorithm A' RETURNs value v. Then the corresponding READ in the constructed execution e of algorithm A also RETURNs value v. We must prove that v is a proper value to RETURN in e'. In e, v is the value of an overlapping WRITE, the value of the last preceding WRITE, or the initial value of A. We consider each possibility in turn. If in e, v is the value of an overlapping WRITE, then WRITE(v) overlaps the original READ in e'. Thus v is a proper value to RETURN in e'. If in e, v is the value of the last preceding WRITE, then either there is a corresponding WRITE(v) in e' or there is not a corresponding WRITE(v) in e' (so no WRITE precedes the READ in e'). If there is a corresponding WRITE(v) in e', then v is a proper value to RETURN in e'. Otherwise v is v_0, the initial value for A'; thus v is a proper value to RETURN in e'. If in e, v is the initial value of A and no WRITE precedes the READ, then the initial value of A is also v_0 and the READ in e' has no preceding WRITE. Thus v is a proper value to RETURN in e'. Therefore algorithm A' satisfies the regular property.

A' trivially satisfies the symmetric property because A satisfies the symmetric property, and $R_{A'}(k-1) \leq m - p$. ■

Lemma 4.4 $R(k-1) \leq R(k) - 1$.

Proof Choose any $A \in SYM(k)$ with $R_A(k) = R(k) = m$. Let C be a reachable configuration of A. Since A is a one-write algorithm, C has a neighbor D such that $f_A(D) \neq f_A(C)$. By Lemma 4.3 with $p = 1$, there exists an $A' \in SYM(k-1)$ with $R_{A'}(k-1) \leq m-1$. Thus $R(k-1) \leq m-1$. ■

Lemma 4.5 *If k is not a power of 2, then $R(k-1) \leq R(k) - 2$.*

Proof Choose any $A \in SYM(k)$ with k not a power of 2 and $R_A(k) = R(k) = m$. If we can show that there exists a reachable configuration C and some $w \neq f_A(C)$ with at least two neighbors D_1 and D_2 such that $f_A(D_1) = f_A(D_2) = w$, then the result would follow from Lemma 4.3, substituting 2 for p. The rest of this proof is devoted to showing that such a configuration exists. Suppose in contradiction that for every reachable configuration C and every $w \neq f_A(C)$, C has at most one neighbor D with $f_A(D) = w$.

Claim 4.1 *For any reachable C, f_A maps all nonreachable neighbors of C to $f_A(C)$.*

> **Proof** Suppose in contradiction that C has one nonreachable neighbor E such that $f_A(E) \neq f_A(C)$. C already has a reachable neighbor D with $f_A(D) = f_A(E)$ because A is a one-write algorithm. This means that C has at least two neighbors mapped by f_A to $f_A(E)$, a contradiction.
>
> **End of Claim**

Claim 4.2 *All configurations are reachable.*

> **Proof** Suppose in contradiction that there exists a nonreachable configuration. Then there exists a reachable configuration C_0 that has a nonreachable neighbor D_0. $f_A(D_0) = f_A(C_0)$ by Claim 4.1. Suppose D_0 and C_0 differ only in bit i. Since we are assuming that the minimum number of binary regular registers is used, there exists some reachable configuration E such that E and C_0 differ in bit i and bit i labels the last edge in some path of algorithm edges in H_A connecting C_0 and E. The length of the path from C_0 to E must be at least 2. Let the path be

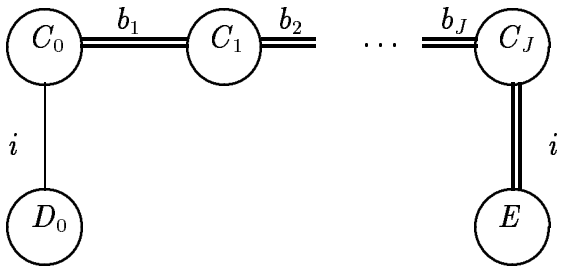

Figure 4.12: Relationships Among the Configurations in the Chain from C_0 to E

denoted by the bits that were changed in the path: $b_1, b_2, \ldots, b_J, i$. Suppose the sequence of configurations in the path is $C_0, C_1, C_2, \ldots, C_J, E$. Then C_J and E differ only in bit i. Figure 4.12 shows the relationships among these configurations. Double lines denote algorithm edges. Single lines denote edges which are not algorithm edges. For all j, $1 \leq j \leq J$, let D_j be the neighbor of C_j that differs from C_j in bit i. Notice that D_0 is nonreachable, and $D_J = E$, which is reachable. Since $D_0, D_1, \ldots, D_J = E$ is the sequence of configurations in some path, there exists a j such that D_{j-1} is nonreachable and D_j is reachable. Figure 4.13 shows the relationships among C_{j-1}, C_j, D_{j-1}, and D_j. Dashed lines denote edges which are not known to be algorithm edges. Let $f_A(C_{j-1}) = v_1$. $f_A(C_j) \neq v_1$ because (C_{j-1}, C_j) is an algorithm edge. Since D_{j-1} is unreachable, $f_A(D_{j-1}) = v_1$ by Claim 4.1. Since D_{j-1} is an unreachable neighbor of reachable D_j, $f_A(D_j) = v_1$ by Claim 4.1. Thus C_j has two neighbors mapped by f_A to v_1, a contradiction.

End of Claim

We proceed by choosing a value from our value set V and counting in two different ways the number of edges of H_A with one endpoint that is mapped by f_A to our chosen value. The results of our two countings must be equal.

Choose some $v \in V$. Let b be the number of configurations C with $f_A(C) = v$. Let B be the set of edges (C, D) such that exactly one of the following is true:

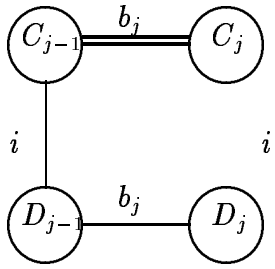

Figure 4.13: Relationships Among C_{j-1}, C_j, D_{j-1}, and D_j

1. $f_A(C) = v$ and $f_A(D) \neq v$.

2. $f_A(C) \neq v$ and $f_A(D) = v$.

For each configuration C such that $f_A(C) = v$, C has $k-1$ neighbors D with $f_A(D) \neq v$ by Claim 4.2 and the assumption made about all reachable configurations. This implies that $|B| = b(k-1)$. For each configuration D such that $f_A(D) \neq v$, D has one neighbor C with $f_A(C) = v$ by Claim 4.2 and the assumption made about all reachable configurations. This implies that $|B| = 2^m - b$. Then $2^m - b = b(k-1)$, which implies that $2^m = kb$, which means that k is a power of 2. This contradicts our assumption that k is not a power of 2. ■

Theorem 4.2 $R(k) > 2k - 2 - \lfloor \log k \rfloor$.

Proof We proceed by induction on k.

Basis: ($k = 4$.) $2k - 2 - \lfloor \log k \rfloor = 4$. By Lemma 4.2, $R(4) > 4$.

Inductive step: ($k > 4$.) Suppose the lemma is true for $k-1$. Now we show that it is true for k. There are two possibilities for k. Either k is a power of 2, or k is not a power of 2.

Case 1: k is a power of 2.

$R(k) \geq R(k-1) + 1$ by Lemma 4.4

$> 2(k-1) - 2 - \lfloor \log(k-1) \rfloor + 1$, by the inductive hypothesis

$= 2k - 2 - 2 - (\lfloor \log k \rfloor - 1) + 1$, because k is a power of 2

$= 2k - 2 - \lfloor \log k \rfloor$.

Case 2: k is not a power of 2.

$R(k) \geq R(k-1) + 2$ by Lemma 4.5

$> 2(k-1) - 2 - \lfloor \log(k-1) \rfloor + 2$, by the inductive hypothesis

$= 2(k-1) - 2 - \lfloor \log k \rfloor + 2$, because k is not a power of 2

$= 2k - 2 - \lfloor \log k \rfloor$. ■

4.1.3 Justifying Restrictions on Readers

In this subsection we justify the restrictions (in the definition of normal form algorithm) that we placed on the readers by showing that general readers do not allow implementations which use fewer physical registers. Theorem 4.3 shows that any one-write algorithm can be converted to a normal form algorithm which uses no more registers. Theorem 4.4 shows that any symmetric algorithm can be converted to a symmetric algorithm using no more registers in which every reader reads each physical register at most once.

Theorem 4.3 *Any one-write algorithm A using m physical registers can be converted to a normal form algorithm A' which uses at most m physical registers.*

Proof The proof uses algorithm transformation techniques as in the proof of Lemma 4.3. Each reader's protocol in algorithm A' is the same as reader 1's protocol in algorithm A, starting in reader 1's initial state, with one exception: the readers in algorithm A' do not perform any physical writes. Instead, they perform writes to local variables (physical register i corresponds to local variable c_i). The writer's protocol in algorithm A' is the same as the writer's protocol in algorithm A. A' is a normal form algorithm. We now prove the regularity of A'. Consider any execution e' of algorithm A'. Let s' be the schedule of e'. We consider each completed READ

r_i in e' in turn. For r_i, we build a sequence of actions, s_i, which will be shown to be a schedule of a possible execution of algorithm A. We obtain s_i from s' by removing all READs (and their associated physical actions) except for r_i and by changing r_i to be a READ by process 1. We now consider each action a_{ij} within r_i. If a_{ij} is a write of a local variable c_l, then we replace a_{ij} with a corresponding physical write to physical register l. If a_{ij} is a read of a local variable c_l, then we replace a_{ij} with a corresponding physical read of physical register l. Otherwise, a_{ij} remains unchanged in s_i. By induction on the number of actions in s_i, we can show that there is an execution e_i of algorithm A with schedule s_i. Suppose r_i in e' RETURNs value v. Then the corresponding READ in execution e_i of algorithm A also RETURNs value v. By the assumption about A, e_i satisfies the regular property. Thus, in e_i, v is the value of an overlapping WRITE, the value of the last preceding WRITE, or the initial value of A (also, of A'). It follows that v is a proper value for r_i to RETURN in e' because the sequences of WRITEs in e_i and e' are the same, and e_i's READ and r_i have the same relationship with the WRITEs. Thus, r_i RETURNs a correct value. Since all completed READs in e' are regular and e' is an arbitrary execution, A' satisfies the regular property. ■

Theorem 4.4 *Any symmetric algorithm A using m physical registers can be converted to a symmetric algorithm A' using at most m physical registers in which every reader reads each physical register at most once.*

Proof The proof uses algorithm transformation techniques as in the proof of Lemma 4.3. By Theorem 4.3, we can assume that A is a normal form algorithm. The writer's protocol in A' is the same as the writer's protocol in A. The reader's protocol in A' is the same as the reader's protocol in A, with the following exception. After a reader in A' reads physical register i for the first time during a READ, it makes a local copy of that register, c_i. It reads the local copy for all subsequent accesses to that physical register during the READ. We now prove the regularity of A'. Consider any execution e' of algorithm A'. We build a corresponding sequence of actions s of algorithm A as follows. We consider each action in e' in turn. If the action is not a read of a local bit c_i by reader j, then the action is placed as is in s. If the action is a read of a local bit c_i by reader j, then we do the following in order:

- place actions to complete the pending WRITE (if there is one)
- determine the sequence of values written to the logical register since the last read of register i by reader j (not including the last WRITE), denoted ws.
- place logical WRITEs (along with the associated physical writes) for those values in $reverse(ws)$
- place a physical read operation of register i by reader j
- place logical WRITEs (along with the associated physical writes) for the values in ws
- handle the last WRITE (making it pending again if it was originally pending).

We can do this easily because A (and A') are symmetric. We can show by induction on the number of actions in s that there exists some execution e of A with schedule s. Suppose a READ by reader j in execution e' of algorithm A' RETURNs value v. Then the corresponding READ in the constructed execution e of algorithm A also RETURNs value v. We must prove that v is a proper value to RETURN in e'. In e, v is the value of an overlapping WRITE, the value of the last preceding WRITE, or the initial value of A (also, of A'). The sets of values written in e and e' which may be RETURNed by the READ are the same (the values may have higher multiplicities in the set of values written in e than in the set of values written in e'), which implies that v is a proper value to RETURN in e'. Thus, algorithm A' satisfies the regular property. ■

4.2 Lower Bounds for Atomicity

In this section we establish lower bounds on the number of registers required by two classes of atomic one-write algorithms. Since atomicity is a stronger property than regularity, we may not be able to transform an arbitrary atomic one-write algorithm to a normal form algorithm. It may help for readers to communicate with each other and the writer by writing to binary registers. Since atomicity implies regularity, Theorems 4.1 and 4.2 are true for normal form atomic one-write algorithms. The proofs for the atomic case are identical to the proofs of Theorems 4.1 and 4.2.

Chapter 5

Must One-Write Algorithms Satisfy the Toggle or Symmetric Properties?

5.1 A One-Write Algorithm Implementing a 3-ary Regular Register but not Satisfying the Toggle Property

Showing that any one-write algorithm would be required to satisfy the toggle property would be a very nice result because it would imply that our algorithm is optimal with respect to space. However, we found an exception to that rule.

We present a one-write algorithm implementing a 3-ary regular register from 3 regular bits which does not satisfy the toggle property. Let the value set of the logical register be $\{a, b, c\}$, and let the initial value of the logical register be a. As usual, a reader reads all 3 bits and returns the value of a function f_{NT} applied to the configuration obtained. The writer changes a bit only when the value of the logical register changes; when the value is changed from v to w, the writer uses the algorithm edges of the 3-dimensional hypercube H_{NT} given in Figure 5.1 and the current configuration of the bits to determine which bit to change. Double lines

in the figure denote algorithm edges. Each reachable configuration C in H_{NT} is labeled with $f_{NT}(C)$. The nodes labeled with d are unreachable configurations; each of these nodes could be mapped by f_{NT} to an arbitrary element of $\{a, b, c\}$. A formal description of the algorithm, called A_{NT}, appears in Figure 5.2. A_{NT} does not satisfy the toggle property because whenever the value of the logical register is changed from a to c or from c to a, either bit 1 or bit 3 is changed, depending on the configurations of the bits.

A_{NT} satisfies the symmetric property. We must now prove that A_{NT} satisfies the regular property.

Theorem 5.1 *Algorithm A_{NT} implements a regular 3-valued register.*

Proof When no write overlaps a read, the read clearly returns the last value written to the register (or the initial value if no value has been written), thus returning a correct value. Now suppose that at least one write overlaps a read. Let C be the configuration read by the reader during execution of A_{NT}. Let S be the set of distinct values written to the register by the last write preceding the read or by some write overlapping the read. If $S = \{a, b, c\}$, then the reader returns a correct value since $f_{NT}(C)$ is in $\{a, b, c\}$. If $|S| = 2$, then we have three cases to consider.

Case 1. $S = \{a, b\}$. Then the actual configurations of the physical registers during the read and after the last write preceding the read must be such that exactly one of the following is true:

1. They are in $\{010, 110\}$. By the code of A_{NT}, $C = 010$ or $C = 110$, implying that $f_{NT}(C)$ is in $\{a, b\}$. Thus the reader returns a correct value.

2. They are in $\{000, 001\}$. By the code of A_{NT}, $C = 000$ or $C = 001$, implying that $f_{NT}(C)$ is in $\{a, b\}$. Thus the reader returns a correct value.

Case 2. $S = \{a, c\}$. Then the actual configurations of the physical registers during the read and after the last write preceding the read must be such that exactly one of the following is true:

1. They are in $\{000, 100\}$. By the code of A_{NT}, $C = 000$ or $C = 100$, implying that $f_{NT}(C)$ is in $\{a, c\}$. Thus the reader returns a correct value.

2. They are in $\{010, 011\}$. By the code of A_{NT}, $C = 010$ or $C = 011$, implying that $f_{NT}(C)$ is in $\{a, c\}$. Thus the reader returns a correct value.

Case 3. $S = \{b, c\}$. Then the actual configurations of the physical registers during the read and after that last write preceding the read must be such that exactly one of the following is true:

1. They are in $\{001, 011\}$. By the code of A_{NT}, $C = 001$ or $C = 011$, implying that $f_{NT}(C)$ is in $\{b, c\}$. Thus the reader returns a correct value.

2. They are in $\{110, 100\}$. By the code of A_{NT}, $C = 110$ or $C = 100$, implying that $f_{NT}(C)$ is in $\{b, c\}$. Thus the reader returns a correct value.

If $|S| = 1$, C is the actual configuration of the physical registers during the read and after the last write preceding the read. Thus $f_{NT}(C)$ is a correct value for the reader to return. ■

5.2 A Nonsymmetric One-Write Algorithm Implementing a 3-ary Regular Register

We present a one-write algorithm implementing a 3-ary regular register from 3 regular bits which does not satisfy the symmetric property. We assume without loss of generality that the value set of the logical register is $\{a, b, c\}$. We also assume that the initial value of the logical register is a. As usual, a reader reads all 3 bits and returns the value of a function f_{NS} applied to the configuration obtained. The writer changes a bit only when the value of the logical register changes; when the value is changed from v to w, the writer uses the algorithm edges of the 3-dimensional hypercube H_{NS} given in Figure 5.3 and the current configuration of the bits to determine which bit to change. Arrows in the figure denote algorithm edges. Each reachable configuration C in H_{NS} is labeled with $f_{NS}(C)$. The node

labeled with d is an unreachable configuration; this node could be mapped by f_{NS} to an arbitrary element of $\{a, b, c\}$. A formal description of the algorithm, called A_{NS}, appears in Figure 5.4.

A_{NS} does not satisfy the symmetric property. The symmetric property is violated twice:

- $(000, 001)$ is an algorithm edge, but $(001, 000)$ is not.
- $(011, 010)$ is an algorithm edge, but $(010, 011)$ is not.

We must now prove that A_{NS} satisfies the regular property.

Theorem 5.2 *Algorithm A_{NS} implements a regular 3-valued register.*

Proof When no write overlaps a read, the read clearly returns the last value written to the register (or the initial value if no value has been written), thus returning a correct value. Suppose that at least one write overlaps a read. Let C be the configuration read by the reader. Let S be the set of values written to the register by the last write preceding the read or by some write overlapping the read. Let CON be the set of actual configurations of the physical registers during the read and after the last write preceding the read. If $S = \{a, b, c\}$, then the reader returns a correct value since $f_{NS}(C)$ is in $\{a, b, c\}$. If $|S| = 2$, then we have three cases to consider.

Case 1. $S = \{a, b\}$. By the code of A_{NS}, $CON = \{000, 010\}$, $CON = \{011, 010, 000\}$, or $CON = \{010, 000\}$. For all three choices for CON, C is in CON by the code of A_{NS}. Thus the reader returns a correct value because $f_{NS}(C)$ is in $f_{NS}(CON)$. The only choice for CON which requires a more detailed analysis is $CON = \{011, 010, 000\}$. If $CON = \{011, 010, 000\}$, then by the code of A_{NS}, 011 must have been the actual configuration of the physical registers after the last write preceding the read. By the code of A_{NS}, $C = 011$ or $C = 010$ or $C = 000$, implying that $f_{NS}(C)$ is in $\{a, b\}$. Thus the reader returns a correct value.

Case 2. $S = \{a, c\}$. By the code of A_{NS}, $CON = \{100, 110\}$, $CON = \{000, 001, 011\}$, or $CON = \{001, 011\}$. For all three choices for CON, C is in

CON by the code of A_{NS}. Thus the reader returns a correct value because $f_{NS}(C)$ is in $f_{NS}(CON)$. The only choice for CON which requires a more detailed analysis is $CON = \{000, 001, 011\}$. If $CON = \{000, 001, 011\}$, then by the code of A_{NS}, 000 must have been the actual configuration of the physical registers after the last write preceding the read. By the code of A_{NS}, $C = 000$ or $C = 001$ or $C = 011$, implying that $f_{NS}(C)$ is in $\{a, c\}$. Thus the reader returns a correct value.

Case 3. $S = \{b, c\}$. By the code of A_{NS}, $CON = \{010, 110\}$ or $CON = \{001, 101\}$. For both choices for CON, C is in CON by the code of A_{NS}. Thus the reader returns a correct value because $f_{NS}(C)$ is in $f_{NS}(CON)$.

If $|S| = 1$, then $|CON| = 1$ by the code of A_{NS}. Thus, C is in CON by the code of A_{NS}. The reader returns a correct value because $f_{NS}(C)$ is in $f_{NS}(CON)$. ■

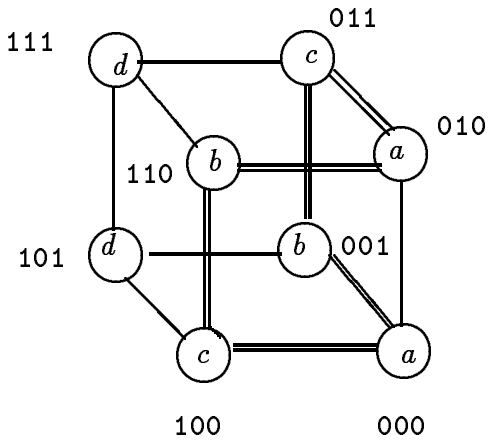

Figure 5.1: The Hypercube H_{NT} for the Non-Toggle Algorithm

Physical Registers (Bits): X_1, X_2, X_3, initially $X_j = 0$, for all $j \in \{1,2,3\}$

Reader i, $1 \leq i \leq n$: variables x_1, x_2, x_3

READ(i):

for $j := 1$ to 3 do

read$_j(i)$

return$_j(i, x_j)$

endfor

RETURN($i, f_{NT}(x_1 \ldots x_3)$)

Writer: variables x_1, x_2, x_3, initially $x_j = 0$, for all $j \in \{1,2,3\}$, and old, initially $old = v_0$

WRITE(v):

if $v \neq old$ then

$i :=$ the bit labeling the algorithm edge between node $x_1x_2x_3$ and its neighbor $y_1y_2y_3$ with $f_{NT}(y_1 \ldots y_3) = v$

write$_i(\overline{x_i})$

ack$_i$

$x_i := \overline{x_i}$

$old := v$

endif

ACK

Figure 5.2: One-Write Algorithm A_{NT}

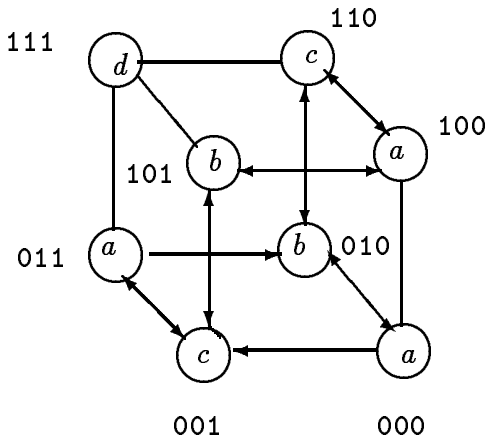

Figure 5.3: The Hypercube H_{NS} for the Nonsymmetric Algorithm

Physical Registers (Bits): X_1, X_2, X_3, initially $X_j = 0$, for all $j \in \{1,2,3\}$

Reader i, $1 \leq i \leq n$: variables x_1, x_2, x_3

READ(i):

for $j := 1$ to 3 do

read$_j(i)$

return$_j(i, x_j)$

endfor

RETURN($i, f_{NS}(x_1 \ldots x_3)$)

Writer: variables x_1, x_2, x_3, initially $x_j = 0$, for all $j \in \{1,2,3\}$, and old, initially $old = v_0$

WRITE(v):

if $v \neq old$ then

$i :=$ the bit labeling the algorithm edge between node $x_1x_2x_3$ and its neighbor $y_1y_2y_3$ with $f_{NS}(y_1 \ldots y_3) = v$

write$_i(\overline{x_i})$

ack$_i$

$x_i := \overline{x_i}$

$old := v$

endif

ACK

Figure 5.4: One-Write Algorithm A_{NS}

Chapter 6

Conclusions from Our Study of Registers

We have proven the existence of a one-write algorithm for implementing a k-ary regular register from binary regular registers. The same algorithm implements a k-ary atomic register from binary atomic registers. The algorithm we have developed uses $k(k-1)/2$ binary registers. It is optimal in the number of binary registers used with respect to all one-write algorithms satisfying the toggle property. We have also improved the lower bound on the number of binary registers required for all one-write algorithms satisfying the symmetric property from k to $2k - 1 - \lfloor \log k \rfloor$. Our lower bound proofs are modular, and they use our general technique for "fooling the reader". We have also simplified the readers and have justified the simplifications. An interesting open question is to determine tight bounds on the number of physical registers needed for symmetric algorithms and more general types of algorithms. By brute force, we believe that we can prove that at least $k + 1$ registers are needed for general regular algorithms. Lemma 4.3, which is our general algorithm transformation technique, may help in obtaining tighter bounds. For example, if one can establish that $p = \Theta(\log k)$, then one can obtain a lower bound of $\Omega(k \log k)$ registers.

Chapter 7

Virtual Shared Objects

7.1 Introduction

In concurrent systems, processes need to share information with each other. Because of software engineering concerns, using shared data objects is a popular method for organizing this information. These shared objects may come from arbitrary abstract data types. However, processes may not have access to a physical shared memory because they may be running on computers in different locations separated by long distances or their computer architecture may not provide a physical shared memory. They must simulate a physical shared memory by using a virtual shared memory.

A consistency guarantee tells the processes using the virtual shared objects what they can expect about the values returned as results of applying operations, even when operations are executed concurrently on the same virtual object. Researchers have defined many types of consistency guarantees, with some strong and some weak and some in between. A strong guarantee is at least as expensive to implement as a weaker guarantee, but it may be impossible (or very hard) to solve a problem by using virtual shared objects providing a weaker guarantee.

Sequential consistency and linearizability are two strong consistency guarantees. They ensure that operations appear to have executed atomically in some sequential order that reflects the order in which operations were executed at each process.

In addition, linearizability ensures that that this sequential order preserves the relative ordering of all non-overlapping operations, even if they were executed by different processes. Sequential consistency is a very popular consistency guarantee in various contexts, including virtual shared memories and multiprocessor caches ([ABM93, AW94]). However, linearizability is a stronger, more intuitive guarantee because sequential consistency provides no clues about the relative ordering of non-overlapping operations performed by different processes. Also, a collection of linearizable objects can be built incrementally because linearizability satisfies the locality property (if individual objects provide a guarantee, then the collective group of objects provides the guarantee) [HW90]. In contrast, a collection of sequentially consistent objects cannot be built incrementally because sequential consistency does not satisfy the locality property [AF92]. Although linearizability is more powerful than sequential consistency, it is still interesting to study both guarantees because the difference between their definitions is very slight.

Because intuition led them to believe that weaker consistency guarantees may be cheaper than strong guarantees, Attiya and Friedman [AF92] formally defined a consistency guarantee called hybrid consistency which systematically weakens either sequential consistency or linearizability; it generalizes guarantees proposed in the computer architecture community ([AH90, DSB88, GLL+90, GMG91]). In hybrid consistency, there are two forms of operations, *strong* and *weak*. Strong operations appear to satisfy a strong consistency guarantee, while weak operations can appear to execute in different orders at different processes as long as the relative ordering of weak and strong operations executed by the same process is preserved. [AF92] showed that hybrid consistency was still strong enough to be computationally useful, proving that it was a consistency guarantee for shared read/write objects that could be used to solve the non-cooperative mutual exclusion problem. [AF92] also showed that read/write objects with some weaker consistency guarantees could not be used to solve the non-cooperative mutual exclusion problem.

It is important to understand the costs of providing these consistency guarantees, both weak and strong, in order to help system designers choose the consistency guarantee which best suits the needs of their applications and is cost-effective. We focus on (distributed) message-passing implementations because they are scalable.

Each process has its own local memory, and all processes run a protocol to provide the illusion of a physical shared memory with a given consistency guarantee.

Attiya and Welch [AW91, Att91, AW94][1] made a comparative study of the costs of implementing sequential consistency and linearizability for basic read/write objects, queues, and stacks. They measured the worst-case time for operations to complete, giving upper and lower bounds to show that the amount of synchrony among the processes using the objects caused the cost difference between sequential consistency and linearizability to vary. With perfect synchrony, their costs are equal. However, sequential consistency is cheaper when processes are only approximately synchronized. Mavronicolas and Roth [MR92] continued their comparative study, improving some of their lower bounds and giving a distributed implementation of linearizable read/write objects in a system with only approximately synchronized processes. Attiya and Friedman [AF92] compared the cost of implementing hybrid consistency with the costs of implementing sequential consistency and linearizability for read/write objects, showing that hybrid consistency is cheaper when mostly weak operations are executed.

Instead of concentrating on specific data types, as the previously described work has done, we study the worst-case response times for operations of *arbitrary* abstract data types in sequentially consistent, linearizable, and hybrid consistent implementations. We show that algebraic properties of the operations of an abstract data type are sufficient for proving many lower bounds and some upper bounds on the worst-case response times. Our work generalizes and unifies previously known results [AW94, MR92, AF92]. As a consequence, we provide specific results about other abstract data types that were not previously considered, such as dictionary sets, cyclic arrays, and read/modify/write objects. Some of these algebraic properties that we used were defined by Weihl, who used them to study concurrency control and recovery in transaction systems [Wei93]. Using Weihl's definitions as a basis helped us to identify interesting new algebraic properties of the operations of abstract data types.

We now give a brief description of our results.

[1] [AW94] subsumes the results of [AW91] and [Att91], so we will just refer to [AW94] in the remainder of this work.

Let d be the maximum message delay of a system of processes connected by a network, and u be the uncertainty in the message delay ($0 \leq u < d$). This implies that the actual message delay may vary between $d - u$ and d.

In Chapters 8 and 9, we compare the costs of implementing sequential consistency and linearizability, varying the amount of synchrony among the processes as in [AW94].

In Chapter 8, we assume that processes have perfectly synchronized clocks and constant message delays ($u = 0$). Any lower bounds proved for such systems will automatically hold in systems with weaker, more realistic timing assumptions.

We considered interactions between and among operations in order to determine the worst-case completion times for the other operations of the abstract data type to be implemented. We found that any pair of (generic) operations that immediately do not commute must collectively take at least d time in any sequentially consistent implementation of objects of its abstract data type. We also found that any pair of (generic) operations that are cyclically dependent must collectively take at least $2d$ time in any sequentially consistent implementation of objects of its abstract data type. Last, but not least, we determined that if a single operation is noninterleavable with an ordered pair of operations, either the single operation must take at least d time or the pair must collectively take at least d time in any sequentially consistent implementation of objects of its abstract data type.

It is often the case that a particular operation or group of operations of an abstract data type is known to be used most frequently. Thus it is desirable to optimize (in terms of worst-case completion time) that operation or group of operations. We have investigated conditions that would allow for this operation or the individual operations of the group to have worst-case completion times of 0 (to be **fast**), meaning that an operation can return immediately based on current local information at its invoking process, in a linearizable implementation of objects from its abstract data type. In our investigation, we have found that the worst-case completion times of operations are not independent. Making one operation fast may require another operation to be **slow** (have a worst-case completion time of $\Omega(d)$).

Operations may be classified as accessors, pure modifiers, self-oblivious operations, immediately self-commuting operations, or none of the above. We give the

relationships among these classes of operations and describe the smallest possible worst-case completion time (fast or slow) of a single operation in these classes in Figure 7.1. All these classes are nonempty.

Our general results are that any single self-oblivious operation can be made fast and that no operation which does not immediately commute with itself can be made fast. We do not know about the smallest possible worst-case completion time for a single operation that is not self-oblivious but does immediately commute with itself. However, a large class of well-known abstract data types has self-oblivious operations. We prove further that any subset of operations consisting only of accessors can be made fast and that any subset of operations consisting only of pure modifiers can be made fast.

What insight do the above results give us about the relative costs of sequential consistency and linearizability? Sequential consistency and linearizability are asymptotically equal in cost for all abstract data types because the worst-case completion times for operations achieved by our linearizable implementations match our lower bound results for sequentially consistent implementations to within constant factors. Additionally, sequential consistency and linearizability are equally costly under two conditions. The first condition is when the operations that can be made fast in linearizable implementations are the most frequently executed operations of their types, because 0 is the smallest possible worst-case completion time for any operation. The second condition is when the total worst-case completion time for all operations in a linearizable implementation matches the sequential consistency lower bound for all operations, causing equal costs for a class of abstract data types when all operations are executed with approximately the same frequency.

In Chapter 9, we consider systems with positive uncertainty in the message delay and approximately synchronized clocks. Our goal is to find a list of algebraic properties causing operations satisfying a property in the list to have positive worst-case completion times ($\Omega(u)$) in linearizable implementations of their abstract data types but allowing them to have worst-case completion times of 0 in sequentially consistent implementations of their abstract data types. This would show that sequential consistency is less expensive than linearizability in this model of process synchrony.

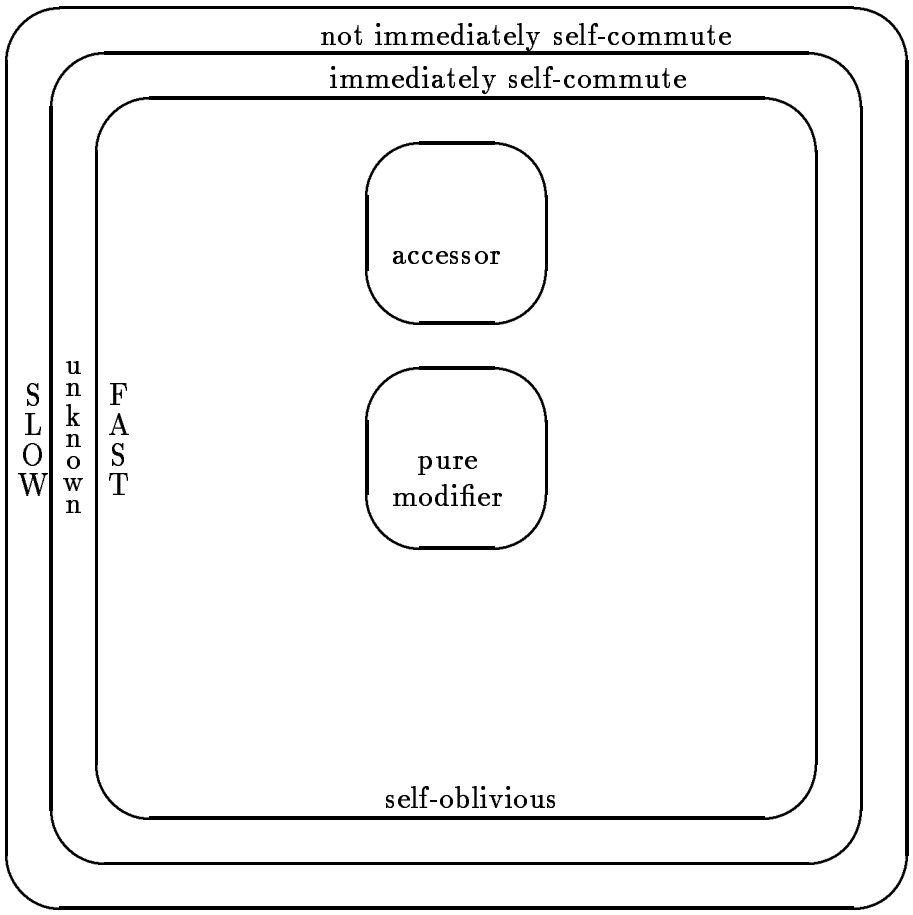

Figure 7.1: Classes of Operations and Their Smallest Worst-Case Completion Times for Sequential Consistency and Linearizability

We have found two properties causing operations to have worst-case completion times that are at least $u/2$ in linearizable implementations of their abstract data types. One of the properties allows an operation satisfying the property to have a worst-case completion time of 0 in a sequentially consistent implementation of its abstract data type, while a restriction of the other property allows an operation satisfying the restriction to have a worst-case completion time of 0 in a sequentially consistent implementation of its abstract data type. These properties are reasonably general, holding for a large class of well-known abstract data types. Under these conditions, sequential consistency is cheaper than linearizability.

In Chapter 10, we consider the weaker consistency guarantee of hybrid consistency. We use the same algebraic properties used in the lower bound proofs for sequential consistency to prove similar lower bounds for hybrid consistent implementations. We use these newly derived lower bounds in conjunction with some upper bounds for implementations providing stronger consistency guarantees to deduce that hybrid consistency is not necessarily cheaper than stronger consistency guarantees. Independently, Friedman [Fri93] compared the cost of implementing hybrid consistency with the costs of implementing sequential consistency and linearizability for read/modify/write objects, queues, and stacks, showing that hybrid consistency is not cheaper, even when mostly weak operations are executed.

We summarize our results and discuss progress towards improving some of them in Chapter 11.

7.2 Definitions

7.2.1 Abstract Data Type Preliminaries

A **sequential specification** [HW90] for an abstract data type contains a set of **operations**, which are ordered pairs of call and response events, and a set of legal operation sequences. The legal sequences of operations reflect the semantics of the abstract data type. For example, for read/write objects, each read operation in a legal sequence returns the value written by the last preceding write operation in the sequence. The call event represents the call for the corresponding operation

from the abstract data type, while the response event represents the value returned by applying the operation. We refer to call events and their matching response events separately because they may not occur atomically (they may be separated in time). For operation op, a call event is of the form $call(arglist)$, where $arglist$ is the argument list (possibly empty) for the operation. For example, a call event for a read operation (respectively, write operation) on a read/write object is of the form $read()$ (respectively, $write(v)$). Similarly, a response event is of the form $resp(retlist)$, where $retlist$ is the list of values returned by the operation and is possibly empty. For example, a response event for a read operation (respectively, write operation) on a read/write object is of the form $ret(v)$ (respectively, $ack()$). We can combine the call and response events to yield $op(arglist)(retlist)$. For instance, we get $read()(v)$ and $write(v)()$ in our running example. We can partition all operations into equivalence classes. All operations with the same name for their call events are in the same **generic operation**. OP_i denotes a generic operation and op_i and op_i^j denote instantiations of OP_i (i.e., non-generic operations).

The notion of sequential specification implicitly assumes some initialization of the object, i.e., $read()(3)$ is legal for a read/write object that has 3 as its initial value. We assume the ability to explicitly initialize an object O using any sequence of operations ρ that is legal for O. Formally, object O_ρ is a **ρ-initialized version of** O if it has the same set of operations as O and σ is legal for O_ρ if and only if $\rho \cdot \sigma$ is legal for O. The assumption of arbitrary explicit initialization is not unreasonable since initialization normally occurs at the beginning of program executions.

We now define two basic algebraic properties of operations that we will use. These definitions come from [Wei93]. Let α and β be operation sequences. α **looks like** β if for every operation sequence γ, $\alpha\gamma$ is legal only if $\beta\gamma$ is legal. If α looks like β, then the user of the abstract data type will never see the result of an operation that allows the user to distinguish β from α after α is executed. If α looks like β and β looks like α, then α and β are **equivalent**. This means that future operations cannot distinguish between α and β.

So far in this subsection, our definitions were stated in terms of sequential operations on a single object. However, we are studying implementations of multiple shared objects in concurrent systems. Thus we need to enhance our notation. If

e is a call event, response event, or whole operation, then $e[O,p]$ denotes that e is performed by process p on object O. If α is a sequence of operations, then $\alpha[O,p]$ denotes that α is performed by the process p on the object O; each entry e of $\alpha[O,p]$ is regarded as $e[O,p]$. A sequence τ of operations for a set of objects is **legal** if, for each object O, the restriction of τ to operations of O, denoted $\tau|O$, is legal for O's abstract data type.

7.2.2 System Model

Our system model is the same as the system model in [AW94].

A **memory consistency system** (mcs) is a set of processes P and a set of clocks $\mathcal{C}$, one for each p in P. Our assumed system consists of a collection of nodes connected by a network. An application program, a real-time clock, and a memory-consistency system (mcs) process are running on each node. The application program asks the mcs process at its node to perform operations on shared objects, and the mcs process returns the results of performing those operations to the application program, by possibly communicating with the other mcs processes in the system with message passing.

An mcs process can read the real-time clock residing at its node. A **clock** is a monotonically increasing function from real time to clock time (both sets are the set of real numbers). A process cannot modify the real-time clock. Processes can only obtain information about time from their clocks.

The following events can occur at the mcs process on node p. We refer later to the mcs process on node p as process p.

- A **call event** occurs when the application program on node p accesses a shared object.
- A **response event** occurs when the mcs process on node p gives a response from a shared object to node p's application program.
- **Message receive events** are of the form receive(p,m,q) for all messages m and all nodes q. A message receive event occurs when the mcs process on node p receives message m from the mcs process on node q.

- **Message send events** are of the form send(p, m, q) for all messages m and all nodes q. A message send event happens when the mcs process on node p sends message m to the mcs process on node q.

- **Timer set events** are of the form timerset(p, T) for all clock times T. This means that p sets a timer to go off when its clock reads T.

- **Timer events** are of the form timer(p, T) for all clock times T. This means that a timer that was set for time T on p's clock goes off.

Call, message receive, and timer events are **interrupt events**.

An **mcs process** (or just process) is an automaton with a set of states, including an initial state, and a transition function. Each interrupt event causes the transition function to be applied. The transition function is a function from states, clock times, and interrupt events to states, sets of response events, sets of message send events, and sets of timer set events (for future clock times). This means that the transition function takes as input the current state, clock time, and interrupt event, and produces a new state, a set of response events for the application process, a set of messages to be sent, and a set of timers to be set for the future.

A **step** of p is a tuple (s, T, i, s', R, M, S), where s and s' are states (s is the current state, and s' is the new state), T is a clock time, i is an interrupt event, R is a set of response events, M is a set of message send events, S is a set of timer set events, and s', R, M, and S are the results of p's transition function acting on s, T, and i.

A **history** of a process p with clock C is a countable sequence of steps such that

- Steps are ordered by T, their time components, in increasing order.

- The old state in the first step is p's initial state.

- The old state of each subsequent step is the new state of the previous step.

- For the subsequence of steps with time component $T = t$, all non-timer events are ordered before any timer event and there is at most one timer event.

An **execution** of an mcs is a set of histories, one for each process p in P with clock C_p in $\mathcal{C}$ which satisfies the following two conditions. First, for all pairs of processes p and q, every message sent from p to q is received by q and every message received by q from p was actually sent by p (reliable message transmission and no duplicated messages). We use this one-to-one correspondence to define the **delay** of any message in an execution to be the difference between the real time of receipt and the real time of sending. Second, a timer is received by p at clock time T if and only if p has previously set a timer for T.

An execution ρ is **admissible** if the following are true:

- For every p and q, every message in ρ from p to q has delay in the range $[d-u, d]$, for fixed nonnegative integers d and u, $u \leq d$.
- For every p, at most one call at p is pending (lacks a matching response) at any given time.

7.2.3 Correctness Conditions

Our definitions are identical to the definitions in [AW94] and [AF92].

Given an execution σ, let $ops(\sigma)$ be the sequence of call and response events appearing in σ in real-time order. We need to specify a tie-breaking mechanism for ordering events which occur at the same real time t. In this ordering, the first group of events is formed by the response events which happen at time t and have their matching call events happening before time t, ordered by their process identifiers. The second group of events in the ordering is formed by the call events which happen at time t and have their matching response events happening at time t, ordered by their process identifiers with a call event immediately preceding its matching response event. The third group of events in the ordering is formed by the call events which happen at time t and have their matching response events happening after time t, ordered by their process identifiers.

We now define the three correctness conditions we study: sequential consistency, linearizability, and hybrid consistency. These definitions of these conditions all