


alized data race free program runs on alpha consistent memories as if they were sequentially consistent. We present a general method for transforming a non-cooperative mutual exclusion algorithm based on sequential consistency into an algorithm based on alpha consistency. We also show that for two well-known mutual exclusion algorithms, a specific analysis of the code provides a more efficient algorithm.

Gharachorloo, Horning, Saxe, Sites and Yu are currently working on further formalization of the Alpha memory model [18]. Given such a formalization it would be interesting to verify that it is indeed captured by alpha consistency.

An important question left open by our work is how to transform programs written for sequentially consistent memories to work under alpha consistency with minimum amount of synchronization, i.e., using a minimal number of MB operations. Since this problem is probably quite hard, it might be more realistic to concentrate on methods that approximate this goal.

The PowerPC is another new microprocessor architecture which includes a specification of the shared memory model for PowerPC based multiprocessors. A formal definition of this model using axioms appears in [12], where it is shown that if all accesses to shared data are protected by lock/unlock then a PowerPC based multiprocessor behaves as if it was sequentially consistent. A natural question is whether this result holds for any kind of release and acquire sequences. It is also interesting to check whether there is a simple way of transforming mutual exclusion algorithms based on sequential consistency to PowerPC based multiprocessors.

Acknowledgements: Marc Snir suggested the Alpha as a test case for our framework; this paper would have never been written if it was not for his suggestion. We would like to thank Kourosh Gharachorloo, Maurice Herlihy and Burkhard Neidecker-Lutz for helping us understand the Alpha memory model. Thanks also to Jennifer Welch and Bill Weihl for helpful comments.

References


of this algorithm. Each processor advances in a complete binary tree of depth \( \log n \). The code of each level \( i > 1 \) can be viewed as part of the critical section for level \( i-1 \); in each level, every pair of processors that calculate the same value for the variable \( b \) in Figure 7 execute the entry and exit section of Peterson’s algorithm for two processors. The proof of correctness for the algorithm in Figure 7 can be generalized for the algorithm in Figure 8 using this observation; the details are omitted.

6 Discussion

The common wisdom in recent years is that multiprocessors based on fast commercial microprocessors provide a better cost/performance ratio than traditional supercomputers. In order to support this trend, the major architectural issues for planning multiprocessors has to be taken into account in the design of new microprocessors, including the definition of the shared memory model. Alpha is a recent example of such a microprocessor. However, the shared memory model of Alpha is described in a framework that is more comfortable for architects and hardware designers than for programmers and software users.

Our definition of alpha consistency attempts to provide programmers and theoreticians with a more convenient definition of the shared memory model of Alpha based multiprocessors. Alpha consistency is used to investigate programming techniques for Alpha based multiprocessors. These techniques allow programmers to develop programs assuming sequentially consistent hardware and run them on alpha consistent hardware. We show that every gener-
Assume, by way of contradiction, that the algorithm does not guarantee logical mutual exclusion. Thus, operations belonging to a critical section of \( p_0 \) overlap with operations belonging to a critical section of \( p_1 \) in \( \tau \). Let \( s_0(\text{need}[0],\text{true}) \) and \( s_0(\text{turn},1) \) be the last stores by \( p_0 \) before entering this critical section. Similarly, let \( s_1(\text{need}[1],\text{true}) \) and \( s_1(\text{turn},0) \) be the last stores by \( p_1 \) before entering its corresponding critical section. Assume, without loss of generality, that \( s_0(\text{turn},1) \overset{\tau}{\rightarrow} s_1(\text{turn},0) \). However, by the assumption that \( p_1 \) started executing its critical section and the requirements from \( \tau \) in Definition 3.1, either 
\[
\begin{align*}
\text{let } s_0(\text{turn},1) & \rightarrow s_1(\text{turn},0) \rightarrow l_1(\text{turn},1) \rightarrow s_0(\text{need}[0],\text{false}),
\end{align*}
\]
Hence, either \( s_1(\text{turn},0) \) is the last store to \( \text{turn} \) before a load that returns \( 1 \) or \( s_0(\text{need}[0],\text{true}) \) is the last store to \( \text{need}[0] \) before a load that returns false. A contradiction to the assumption that \( \tau \) is legal.

Lemma 5.8 The algorithm in Figure 7 is deadlock free based on alpha consistency.

Proof: Assume, by way of contradiction, that there exists an infinite run \( \sigma \) that has a deadlock and consider the sequence of operations \( \tau \) as guaranteed by Definition 3.1. Assume, without loss of generality, that \( p_0 \) executes the entry section from some point on in \( \tau \), but \( p_1 \) executes its critical section only a finite number of times in \( \tau \).

Let \( s_1(\text{need}[1],\text{false}) \) be the store in the last exit section executed by \( p_1 \) in \( \tau \). Since we assumed that \( p_0 \) executes the entry section from some point on in \( \tau \), \( \tau \) includes an infinite number of operations of the form \( l_0^\dagger(\text{need}[1],u_k) \). Thus, there exists a load \( l_0^\dagger(\text{need}[1],u_k) \) such that \( s_1(\text{need}[1],\text{false}) \overset{\tau}{\rightarrow} l_0^\dagger(\text{need}[1],u_k) \). Since \( \tau \) is legal and since \( s_1(\text{need}[1],\text{false}) \) is the last store to \( \text{need}[1] \) in \( \tau \), \( u_k = \text{true} \). Hence, \( p_0 \) should have entered its critical section following \( l_0^\dagger(\text{need}[1],u_k) \) and the corresponding \( l_0(\text{turn},v) \). A contradiction.

Lemma 5.9 The algorithm in Figure 7 is starvation free based on alpha consistency.

Proof: Assume, by way of contradiction, that there exists an infinite run \( \sigma \) that has starvation. Assume, without loss of generality, that \( p_1 \) is executing the entry section from some point on in \( \tau \), but never executes the corresponding critical section. By Lemma 5.8, \( p_1 \) enters and leaves its critical section infinitely often in \( \tau \). Thus, there exists a store \( s_1^\dagger(\text{turn},0) \) such that \( s_0(\text{need}[0],\text{true}) \overset{\tau}{\rightarrow} s_0(\text{turn},1) \overset{\tau}{\rightarrow} s_1^\dagger(\text{turn},0) \). However, since we assumed that \( p_1 \) enters its critical section, either the following load from \( \text{turn} \) by \( p_1 \) returns \( 1 \), or the following load from \( \text{need}[0] \) returns false. Thus, either \( s_1(\text{turn},0) \) is the last store to \( \text{turn} \) that appears in \( \tau \) before a load from \( \text{turn} \) that returns \( 1 \), or \( s_0(\text{need}[0],\text{true}) \) is the last store to \( \text{need}[0] \) that appears in \( \tau \) before a load from \( \text{need}[0] \) that returns false. Hence, \( \tau \) is not legal. A contradiction.

Figure 8 presents a generalization of the algorithm in Figure 7 for \( n \) processors [28]. Note that Lines 4–8 are the entry section of the algorithm in Figure 7 and Line 12 is the exit section.
Figure 7: Peterson’s algorithm using alpha consistency – code for processor $p_i$, $i = 0, 1$.

By Lemma 5.5, there exists another processor $p_k$ that enters and leaves its critical section infinitely often in $\tau$. Let $s_l^1(\text{number}[i], v)$ be the last store in $\tau$ to $\text{number}[i]$. Thus, there exists a load $l_k^1(\text{number}[i], u)$ that is invoked in Line 3 which is ordered in $\tau$ after $s_l^1(\text{number}[i], v)$. By the legality of $\tau$, $u = v$. Let $s_l^2(\text{number}[i], w)$ be the store to $\text{number}[k]$ that follows $l_k^1(\text{number}[i], u)$ in $\tau$. Thus, $w > u = v$. Let $l_k^3(\text{number}[i], z)$ be the next load by $p_k$ from $\text{number}[i]$. Hence, $z = u < w$. A contradiction to the assumption that $p_k$ enters and leaves its critical section infinitely often in $\tau$.

5.4 Peterson’s Algorithm

Figure 7 presents Peterson’s mutual exclusion algorithm [29] for two processors with the addition of several MB operations that guarantee its correctness under alpha consistency. Lines 1–5 are the entry section; Lines 6–7 are the exit section. Note that the algorithm in Figure 7 uses less MB operations than required by the general method: One MB operation is saved at the end of the exit section. Even more significant is the fact that there is no MB operation between the two loads in Line 5 which is likely to be executed many times during every execution of the entry section. In the code we use the notation $l(x) = v$ to denote a load of $x$ returning the value $v$, for any read/write object $x$ and value $v$.

Lemma 5.7 The algorithm in Figure 7 guarantees logical mutual exclusion based on alpha consistency.

Proof: In order to prove mutual exclusion we have to show that operations that belong to critical sections appear in $\tau$ in a non-overlapping manner. Note that due to the control operation immediately before entering the critical section and the MB operation immediately after leaving the critical section, operations that belong to one critical section do not overlap with operations that belong to another critical section by the same processor.
By interchanging the roles of $i$ and $j$ in the proof of Claim 5.4, we can show that $(u_i, i) > (u_j, j)$. This contradicts Claim 5.4.

Lemma 5.5 The algorithm in Figure 6 is deadlock free based on alpha consistency.

Proof: Assume, by way of contradiction, that there exists a run $\sigma$ of the algorithm in Figure 6 that contains a deadlock, and let $\tau$ be a sequence of operations from the definition of alpha consistency in which the deadlock occurs. We say that processor $p_i$ is *deadly* if $p_i$ is executing the entry section from some point on in $\tau$, but there is no processor that enters the critical section following this point. Denote by $u_j$ the last value stored in $\tau$ by $p_i$ to $\text{number}[j]$ due to the execution of Line 4, for any $j$ (if such a store exists). Let $p_i$ be the processor for which the value of $(u_i, i)$ is minimal among the deadly processors. Thus, either $p_i$ continually executes Line 10, reading $\text{doorway}[k]$ for some $k$, or $p_i$ continually executes Lines 11–12, reading $\text{number}[k]$ for some $k$.

If $p_i$ continually executes Line 10, then since no processor enters its critical section, there is a prefix of $\tau$ that includes all operations that are invoked by $p_i$ while executing Lines 1–6. Let $s_k^1(\text{doorway}[k], \text{true})$ and $s_k^2(\text{doorway}[k], \text{false})$ be the last stores to $\text{doorway}[k]$. Since there is an MB instruction in the program between these two stores, $s_k^1(\text{doorway}[k], \text{true}) \xrightarrow{MB_k} s_k^2(\text{doorway}[k], \text{false})$. Since there are infinite number of loads by $p_i$ from $\text{doorway}[k]$, there exists a load $l_i(\text{doorway}[k], z)$ such that $s_k^2(\text{doorway}[k], \text{false}) \xrightarrow{l_i} l_i(\text{doorway}[k], z)$. Since $\tau$ is legal and $s_k^2(\text{doorway}[k], \text{false})$ is the last store to $\text{doorway}[k]$, $z = \text{false}$. A contradiction to the assumption that $p_i$ continually executes Line 10, reading $\text{doorway}[k]$.

Assume that $p_i$ continually executes Lines 11–12, reading $\text{number}[k]$ for some $k$. By the code of the algorithm, this can only happen if the value returned from $\text{number}[k]$ is greater than 0, meaning that there exists a prefix of $\tau$ in which $p_k$ is executing the entry section. By the minimality of $(u_i, i)$ and the code in Line 12, following some prefix of $\tau$, $p_k$ has left its critical section and never tries to enter it again. Let $s_k^1(\text{number}[k], 0)$ be the last store by $p_k$ to $\text{number}[k]$. Due to the MB instruction in Line 13, $s_k^1(\text{number}[k], 0)$ is the last store to $\text{number}[k]$ in $\tau$. Since there are infinite number of loads by $p_i$ from $\text{number}[k]$, there exists a load $l_i(\text{number}[k], z)$ such that $s_k^1(\text{number}[k], 0) \xrightarrow{l_i} l_i(\text{number}[k], z)$. Since $\tau$ is legal and $s_k^1(\text{number}[k], 0)$ is the last store to $\text{number}[k]$, $z = 0$. A contradiction to the assumption that $p_i$ continually executes Lines 11–12 reading $\text{number}[k]$.

Lemma 5.6 The algorithm in Figure 6 is starvation free based on alpha consistency.

Proof: Assume, by way of contradiction, that there exists a run $\sigma$ of the algorithm in Figure 6 in which some process $p_i$ is starved, and let $\tau$ be the sequence of operations from the definition of alpha consistency in which the starvation occurs. By assumption, $p_i$ continually executes the entry section from some point on in $\tau$, but never executes the corresponding critical section in $\tau$. Thus, either $p_i$ continually executes Line 10 or $p_i$ continually executes Lines 11–12 in $\tau$. 

22
1: \( s(\text{doorway}[i], \text{true}) \);
2: \( M B \);
3: \( \text{my\_num} := 1 + \max\{\text{number}[j] \mid j \neq i\} \);
4: \( s(\text{number}[i], \text{my\_num}) \);
5: \( M B \);
6: \( s(\text{doorway}[i], \text{false}) \);
7: \( M B \);
8: for \( i := 1 \) to \( n \) do
9: \hspace{1em} if \( i \neq j \) then
10: \hspace{2em} if \( i(\text{doorway}[j]) \) then goto 11;
11: \hspace{2em} other := l(\text{number}[j]);
12: \hspace{2em} if \( (\text{other} \neq 0) \) and \( (\text{other}, j) < (\text{my\_num}, i) \) then goto 12;
\hspace{1em} <critical section>
13: \( MB \);
14: \( s(\text{number}[i], 0) \);
\hspace{1em} <remainder section>

Figure 6: Lamport’s bakery algorithm using alpha consistency – code for processor \( p_i, i = 1, \ldots, n \).

previous load from \( \text{number}[j] \) and \( l^{k_i}_j(\text{number}[j], z_i) \) be the last load from \( \text{number}[j] \) before \( p_i \) enters \( CS_i \). Hence, \( u_i \) is the last value written to \( \text{number}[i] \) during \( \tau \); \( v_j \) is the last value read by \( p_i \) from \( \text{number}[j] \) before \( u_i \) is calculated (Line 3 in the code); \( z_j \) is the last value read by \( p_i \) from \( \text{number}[j] \) before entering \( CS_i \) (Line 11 in the code). We define \( s^{k_i}_j(\text{number}[j], u_j) \), \( l^{k_i}_j(\text{number}[i], v_j) \) and \( l^{k_i}_j(\text{number}[j], z_j) \), and in particular \( u_j, v_j \) and \( z_j \), similarly for \( p_j \).

Note that by the code of Line 3, \( \text{my\_num} \) is always greater than 0. Since the instructions in Line 3 and Line 4 touch the same local register, the following claim is immediate:

Claim 5.3 \( u_i > 0 \) and \( u_j > 0 \).

Claim 5.4 \((u_j, j) > (u_i, i)\).

Proof: Since \( p_i \) is executing the critical section, then by the code of the algorithm either \((z_j, j) > (u_i, i)\) or \( z_j = 0 \). If \((z_j, j) > (u_i, i)\), then by the legality of \( \tau \), \( u_j \geq z_j \). Hence, \((u_j, j) > (u_i, i)\).

If \( z_j = 0 \), then by the legality of \( \tau \), \( s^{k_i}_j(\text{number}[i], u_i) \xrightarrow{\tau} l^{k_i}_i(\text{number}[j], z_i) \xrightarrow{\tau} s^{k_i}_j(\text{number}[j], u_j) \). Thus, the stores and loads from \( \text{doorway}[j] \) guarantee that \( s^{k_i}_j(\text{number}[i], u_i) \xrightarrow{\tau} l^{k_i}_i(\text{number}[j], v_j) \xrightarrow{\tau} s^{k_i}_j(\text{number}[j], u_j) \).
Hence, by the legality of \( \tau \), \( u_j > u_i \).
control instruction between every two memory instructions of the same processor that touch
different local registers and access different locations is sequentially consistent.

Recall that in every run of \(\mathcal{A}'\), there exists an MB operation immediately before entering
a critical section and immediately after leaving a critical section. Assume there is a violation
of logical mutual exclusion in some sequence \(\tau\) from the definition of alpha consistency. Then
either there is a deadlock in \(\tau\), or there exists a prefix \(\tau''\) of \(\tau\) such that there are at least two
processes executing the critical section together at the end of \(\tau''\).

For an alpha consistent run of \(\mathcal{A}'\), let \(\tau\) be the sequence guaranteed by the definition of
alpha consistency. We then eliminate from \(\tau\) all operations that are not part of an entry or
exit section. Denote the resulting sequence by \(\tau'\). Since \(\mathcal{A}\) is non-cooperative, \(\tau'\) includes all
accesses to every object that is accessed in \(\tau'\). Also, since \(\tau\) is legal, \(\tau'\) is legal too. By the
above discussion, \(\tau'\) is sequentially consistent. Hence, if \(\tau\) violates any of the requirements
of logical mutual exclusion, then so does \(\tau'\). Therefore, we have a sequentially consistent run
of \(\mathcal{A}\) (with empty critical sections) that violates some of the requirements of logical mutual
exclusion. This contradicts the assumption that \(\mathcal{A}\) is a mutual exclusion algorithm based on
sequential consistency.

5.3 Lamport’s Bakery Algorithm

Figure 6 presents Lamport’s mutual exclusion algorithm [22] with several MB operations added
to guarantee its correctness under alpha consistency. Lines 1–12 are the entry section; Lines
13–14 are the exit section. Note that the algorithm in Figure 6 uses less MB operations than
required by the general method: there are no MB operations between the loads from \(\text{number}[j]\)
in Line 3. This saves \(n-2\) MB operations for each execution of the entry section. Also, there is
no MB operation at the beginning of the entry section and at the end of the exit section.
Note the similarities between the proof of the algorithm w.r.t. alpha consistency and the proofs
w.r.t. sequential consistency [8, 22, 30].

**Lemma 5.2** The algorithm in Figure 6 guarantees logical mutual exclusion based on alpha
consistency.

**Proof:** Consider an alpha consistent run of the algorithm in Figure 6 and let \(\tau\) be the
sequence of operations guaranteed by the definition of alpha consistency. Due to the control
operation immediately before entering the critical section and the MB operation immediately
after leaving the critical section, operations belonging to one critical section do not overlap
with operations belonging to another critical section by the same processor. Thus, we need to
show that the critical sections of different processors do not overlap in \(\tau\).

Assume, by way of contradiction, that the operations of two critical sections \(CS_i^j\) and \(CS_k^j\)
overlap in \(\tau\) and let \(\tau'\) be some prefix of \(\tau\) that includes operations of both \(CS_i^j\) and \(CS_k^j\).

For the rest of this proof, let \(\tau'\) be the shortest prefix of \(\tau\) that includes operations of both
\(CS_i^j\) and \(CS_k^j\). Let \(s_i^j(\text{number}[i], u_i)\) be the last store to \(\text{number}[i]\), \(l_k^j(\text{number}[j], v_j)\) be the
A, consider a flow control sequence for processor \( p_i \) and let \( CS^k_i \) be the set of operations invoked by processor \( p_i \) during the \( k \)th time that \( p_i \) executes the critical section in this sequence.

Let \( R \) be an alpha consistent run of \( A \) and let \( \tau \) be a sequence of operations as required in the definition of alpha consistency. Consider the \( CS^k_i \)'s induced by the flow control sequences from the definition of alpha consistency. \( A \) provides \textit{logical mutual exclusion} if for any four operations \( op^1_i, op^2_i \in CS^k_i \) and \( op^1_j, op^2_j \in CS^l_j \), \( op^1_i \xrightarrow{\tau} op^1_j \) if and only if \( op^2_i \xrightarrow{\tau} op^2_j \). This implies that there is a total order on all critical section executions.

Deadlock freedom and starvation freedom are defined as usual, but with respect to \( \tau \). That is, \textit{deadlock freedom} means that if \( \tau \) is infinite and there exists a processor \( p_i \) that is in its entry section from some point on in \( \tau \), then there is another processor \( p_j \) that enters (and leaves) its critical section infinitely often in \( \tau \). \textit{Starvation freedom} means that if \( \tau \) is infinite and there exists a processor \( p_i \) which executes the entry section at some point in \( \tau \), then \( p_i \) executes the critical section at some later point in \( \tau \).

### 5.2 The General Method

A mutual exclusion algorithm is \textit{non-cooperative} if variables that are accessed inside the entry and exit sections are not accessed in the critical and remainder sections (cf. [10]). Let \( A \) be a non-cooperative mutual exclusion algorithm based on sequential consistency. Add to the entry and exit sections of \( A \) the minimal number of MB instructions such that:

1. there is either an MB instruction or a control instruction between every two memory instructions that touch different local registers and access different objects, and
2. the last instruction in both the entry and exit sections as well as the first instruction in the exit section is an MB instruction.

Denote the resulting algorithm by \( A' \). We have:

\textbf{Theorem 5.1} \( A' \) guarantees \textit{logical mutual exclusion based on alpha consistency}.

The proof of this theorem follows closely the lines of the proof of a similar result for hybrid consistency [9, Theorem 5.3]. We outline this proof below.

By the definition of alpha consistency (Definition 3.1), the order of any pair of operations by the same processor such that at least one of them is either an MB operation or a control operation is the same as in the flow control sequence for their invoking processor. Similarly, the order of any pair of operations by the same processor that touch the same local register or access the same memory object is the same as in the flow control sequence for their invoking processor. Thus, every alpha consistent run of a program in which there is either an MB instruction or a
Lemma 4.7 If $\tau$ is not fully admissible, then there exists a prefix of a sequential execution of \textit{Prog} which contains a generalized data race.

The proof of the theorem is now immediate. Let $R$ be an alpha consistent run of a GDRF program \textit{Prog}. Fix a set of flow control sequences $fcS_i$ from the definition of alpha consistency. Let $\tau$ be a minimal legal permutation of the operations in $R$ as guaranteed by the definition of alpha consistency. Since \textit{Prog} is GDRF, no sequential execution has a generalized data race and Lemma 4.7 implies that $\tau$ is fully admissible. Thus, $R$ is a sequentially consistent run of \textit{Prog}.

Remark: It is possible to define a data race to occur between two conflicting operations if they are not properly ordered. Clearly, the proof of Theorem 4.1 can be used for the modified definition, yielding a somewhat stronger result. We have chosen to define data races using release and acquire sequences since this definition is more constructive. Moreover, it seems that it is easier to modify the existing techniques for detecting data races to work with release and acquire sequences rather than with the alternative definition.

5 Programming with Critical Sections

In this section we present a general method that transforms any non-cooperative mutual exclusion algorithm based on sequential consistency into an algorithm based on alpha consistency. Following this, we give two examples in which an analysis of a specific mutual exclusion algorithm yields a more efficient code: Lamport’s bakery algorithm [22] and Peterson’s algorithm [29].

5.1 Logical Mutual Exclusion

The usual definition of mutual exclusion requires that a critical section is not executed by two processors at the same (real) time. However, the definition of alpha consistency (Definition 3.1) describes only the order in which operations are viewed. In particular, this definition has no timing information; operations may be viewed in an order different than the order they occur in real time. Moreover, operations of the same processor may be viewed in an order different than the order implied by the flow control sequence for their invoking processor. Thus, the usual definition of mutual exclusion is not applicable in our framework and we need to redefine the problem. In particular, the fact that operations are executed in a non-overlapping manner in real time does not imply that these operations will be seen by all processors in a non-overlapping manner. We overcome this difficulty by defining \textit{logical mutual exclusion}. A more detailed discussion of logical mutual exclusion appears in [9, 17].

We use the usual division of the code of a mutual exclusion algorithm into four disjoint sections: \textit{entry}, \textit{critical}, \textit{exit} and \textit{remainder} (cf. [30]). Given a mutual exclusion algorithm
2. The last operation in $\pi'$ was added to $\pi''$.

3. $\text{op}_q$ should not be invoked by the evaluation of the of the preceding control operation in $\text{Prog}$.

We now show:

**Lemma 4.6** $\pi''$ contains a pair of operations which are not properly ordered.

**Proof:** If the construction of $\pi''$ terminates due to the first condition then the claim holds immediately.

If the construction of $\pi''$ terminates due to the second condition, then $\pi''$ contains both $\text{op}_j^1$ and $\text{op}_k^2$. If $\text{op}_j^1$ and $\text{op}_k^2$ are not properly ordered, then we are done. Otherwise, since $\text{op}_j^1$ and $\text{op}_k^2$ are not properly ordered in $\pi'$, either $\text{op}_j^1$ or $\text{op}_k^2$ accesses a different object in $\pi''$ than it accesses in $\pi'$. Assume, without loss of generality, that the object accessed by $\text{op}_j^1$ in $\pi''$ is different than the object accessed by $\text{op}_k^2$ in $\pi'$. Hence, there exists a read $r_j^3$ ordered before $\text{op}_j^1$ in $\pi'$ (and hence also in $\pi'$) whose corresponding instruction touches the same local register as the instruction corresponding to $\text{op}_j^1$, and the value returned by $r_j^3$ in $\pi''$ is different than the value returned by $r_j^3$ in $\pi'$. The value returned by $r_j^3$ is different only due to some conflicting write $w_j$ that was inserted to $\pi'$ and does not appear before $r_j^3$ in $\pi'$. Note that $MB_i \xrightarrow{\pi'} w_i$ only if $MB_i \xrightarrow{\pi''} w_i$. This is due to the way $\pi''$ is constructed and the fact that $\pi'$ is a prefix of $\tau$, which obeys the requirements in the definition of alpha consistency. Hence, since $w_i$ is not in $\pi'$ and since only MB operations that appear in $\pi'$ may appear also in $\pi''$, there is no MB operation by $p_i$ ordered after $w_i$ in $\pi''$. In particular, there is no MB operation by $p_i$ ordered between $w_i$ and $r_j^3$ in $\pi''$. Therefore, $w_i$ and $r_j^3$ are not properly ordered in $\pi''$.

If the construction of $\pi''$ terminates due to the last condition then the condition of a control operation $\text{ctrl}_i$ in $\pi''$ is evaluated differently in $\pi'$ than in $\pi''$. Therefore, there exists a read $r_q^3$ that is ordered in $fcs_q$ before $\text{ctrl}_i$, such that the value returned by $r_q^3$ in $\pi''$ is different than the value returned by $r_q^3$ in $\pi'$. The value returned by $r_q^3$ can only be different due to some conflicting write $w_i$ that was inserted to $\pi'$ and does not appear in $\pi'$. Since $w_i$ was not in $\pi'$, there is no MB operation by $p_i$ ordered after $w_i$ in $\pi''$. In particular, there is no MB operation by $p_i$ ordered between $w_i$ and $r_q^3$ in $\pi''$. Therefore, $w_i$ and $r_q^3$ are not properly ordered in $\pi''$. A contradiction to the assumption that there does not exist a pair of conflicting operations which are not properly ordered in $\pi''$. 

It is clear that $\pi''$ is a prefix of a sequential execution of $\text{Prog}$. Thus, we have shown that if $\pi$ is not fully admissible, then there exists a prefix of a sequential execution of $\text{Prog}$, namely $\pi''$, which contains two operations that are not properly ordered.

The next lemma follows immediately from Claim 4.2 and Lemma 4.3.
1. a read that was added is not legal in $\pi'$,
2. a read that became illegal in $\pi'$ due to the insertion of a write,
3. a control instruction that should be evaluated differently due to the insertion of other operations, or
4. an operation accesses an object that it should not or writing a value that it should not due to the fact that local registers may hold different values now.

An operation is *messy* if it satisfies one of the above conditions; if an operation is not messy then it is a correct execution of the appropriate instruction. Note that an operation can be messy only due to the insertion of some operation $op_q$ that was not in $\pi'$. Furthermore, $op_q$ is ordered in $fcs_q$ after the last $MB$ operation and after the last control operation that invoked by $p_q$ that appears in $\pi'$. To complete the proof, we fix $\pi'$ to get a new sequence of operation $\pi''$ which is a prefix of a sequential execution of $Prog$ and contains a pair of operations which are not properly ordered.

In the construction of $\pi''$ we use the following definitions. Consider a sequence of operations $\pi$ and an operation $op$ in $\pi$. A value $v$ is the *true value of an object* $x$ in $\tau$ up to $op$ if $v$ is the value written by the last successful write to $x$ that appears in $\tau$ before $op$, or the initial value of $x$ if there does not exist such a write. A value $v$ is the *true value of a local register* $reg$ in $\tau$ up to $op$ if $v$ is the last value stored into $reg$ in $\tau$ before $op$, or the initial value of $reg$ if no value was stored to $reg$ before $op$ in $\tau$.

The sequence $\pi''$ is constructed by considering the operations of $\pi'$ one by one, in their order of appearance in $\pi'$. We start with an empty sequence. Each operation $op_q$ in $\pi'$ is added to the constructed sequence according to the following conditions:

1. If $op_q$ is not messy, then add $op_q$ to $\pi''$.
2. If $op_q$ is a read $r_q(x; v)$ and $u$ is the true value of $x$ in $\pi''$ up to $r_q(x; v)$, then add $r_q(x; u)$ to $\pi''$. (Since $r_q(x; v)$ is messy, this implies that $v \neq u$.)
3. If the instruction corresponding to $op_q$ is either write($reg_1; \cdot$) or read($reg_1$) and the true value of $reg_1$ in $\pi''$ up to $op_q$ is $y$, then add $op_q$ to $\pi''$ and change the object it accesses to $y$.
4. If the instruction corresponding to $op_q$ is write($\cdot, reg_1$) and the true value of $reg_1$ in $\pi''$ up to $op_q$ is $u$, then add $op_q$ to $\pi''$ and change the value it writes to $u$.

We terminate the construction of $\pi''$ if one of the following conditions holds:

1. There are two operations in $\pi''$ that are not properly ordered. Note that in this case, $op_q$ is one of these operations and it is the last operation in $\pi''$. 

16
2. $op_i^2(x, v)$ and $op_i^1(x, v)$ are not properly ordered in $\tau$, and
3. there is no pair of switched operations in $\tau$ up to $op_i^1(x, w)$.

**Proof:** Since $\tau$ is not fully admissible, there exists at least one pair of switched operations in $\tau$. Let $\langle op_{j}^2(x, v), op_{k}^1(y, u) \rangle$ be the first ordered pair of switched operations, i.e., there is no other pair of switched operations which is completely ordered before $op_{k}^1(y, u)$ in $\tau$. If there is more than one ordered pair that share the same second operation (namely $op_{k}^1$), then choose the pair whose first operation is latest.

Since $\tau$ is minimal, $op_{j}^2(x, v)$ and $op_{k}^1(y, u)$ are switched in order to preserve legality. Note that since the order of every two operations to the same object must be preserved by $\tau$ (the third property in Definition 3.1), $x \neq y$. Therefore, there exists an operation $op_i^1(x, w)$ which conflicts with $op_{j}^2(x, v)$ and $op_{k}^1(x, v)$ $\xrightarrow{sc_{j}} op_{j}^1(x, w) \xrightarrow{sc_{k}} op_{k}^1(y, u)$.

It is also the case that $j \neq k$. Suppose otherwise that $j = k$. Since $op_{j}^2$ and $op_{k}^1$ are conflicting operations then by the third requirement in the definition of alpha consistency (Definition 3.1), $op_{j}^2 \xrightarrow{f \epsilon \Sigma} op_{k}^1$. Thus, $op_{k}^1$ and $op_{j}^2$ are switched and we would have chosen $\langle op_{j}^1(x, w), op_{k}^1(y, u) \rangle$ as the first ordered pair of switched operations. Since $op_{j}^2$ and $op_{j}^1$ are switched, there is no MB operation by $p_j$ between them in $\tau$. Hence, $op_{j}^2$ and $op_{j}^1$ are not properly ordered. The claim follows.

Let $I$ be the set of operations in $\tau$ that influence either $op_{j}^2$ or $op_{k}^1$ (including $op_{j}^2$ and $op_{k}^1$ themselves). Let $\tau'$ be the shortest prefix of $\tau$ that includes every operation in $I$. Let $\pi$ be the subsequence of $\tau'$ which contains exactly the set of all operations in $I$. Since the influence relation is consistent with $\rightarrow$, which in turn is consistent with the flow control sequences, we have:

**Claim 4.5** For every pair of operations $op_i^1$ and $op_i^2$ in $\pi$, if $op_i^1 \xrightarrow{\tau} op_i^2$ then $op_i^1 \xrightarrow{f \epsilon \Sigma} op_i^2$, for every $l \in \{1, \ldots, n\}$.

Now, we construct a new partially admissible (with respect to $\{f \epsilon \Sigma\}_{l=1}^n$) sequence $\pi'$ by adding to $\pi$ every operation $op_i^l$ not in $\pi$, such that $op_i^1 \xrightarrow{f \epsilon \Sigma} op_i^2$ and $op_i^2$ is in $\pi$, for each processor $l$, as follows:

For every such operation that was originally in $\tau'$, add it in the same place as in $\tau'$. For all other operations, add them arbitrarily, maintaining consistency with $\xrightarrow{f \epsilon \Sigma}$, for all $l$. Label every store-conditional $sc_{j}(x, v)$ in $\tau'$ as successful if $sc_{j}(x, v)$ is labeled as successful in $\tau$ and $sc_{j}(x, v)$ appears in $\tau'$ after a load-locked $ll_{j}(x, u)$ such that there are no successful writes by other processes between them; otherwise, label $sc_{j}(x, v)$ as unsuccessful in $\tau'$. By the definition of alpha consistency, none of the added operations (that is not in $\tau'$) is an MB operation.

Note that $\tau'$ may not be a prefix of a sequential execution of $\text{Prog}$, as there may be instructions that are not executed properly. The possible operations are only as follows:
The influence relation is the transitive closure of direct influence. Thus the influence relation is also defined with respect to a set of flow control sequences (and the control orders derived from them). Although we will not usually explicitly mention them, the influence relation will be used with admissible operation sequences and the corresponding flow control sequences (with the control orders derived from them). Note that an operation directly influences another operation only if it is ordered before it in $\tau$ (if $\tau$ is alpha consistent).

We now turn to the proof of the main result of this section. Fix a GDRF program $\text{Prog}$, an alpha consistent run $R$ of $\text{Prog}$ and, as guaranteed by the definition of alpha consistency, a flow control sequence $fcs_i$ for each processor $p_i$.

Let $T$ be the set of all operation sequences $\tau$ that satisfy Definition 3.1 (alpha consistency). For $\tau$ in $T$, an ordered pair of operations of $p_j$, $op_j^1$ and $op_j^2$, is switched in $\tau$ if $op_j^2 \rightleftharpoons op_j^1$ but $op_j^1 \not\rightarrow fcs_j \not\rightarrow op_j^2$.

Let $\tau$ be some element of $T$ with a minimal set of switches. That is, there does not exist $\tau' \in T$, such that the set of pairs of switched operations in $\tau'$ is strictly contained in the set of pairs of switched operations in $\tau$. Thus, in a sequence $\tau \in T$ with a minimal set of switches, a pair of operations is switched only if the "switching" is necessary in order to preserve legality.

**Definition 4.2** A pair of conflicting operations $op_i^1$ and $op_j^2$, $i \neq j$, is properly ordered in a sequence of operations $\tau$ if there exist two operations, $MB_i$ and $MB_j$, such that $op_i^1 \rightleftharpoons MB_i \rightarrow MB_j \rightarrow op_j^2$. (It is possible that $op_i^1 = MB_i$ and/or $op_j^2 = MB_j$.)

The following claim is immediate:

**Claim 4.2** For every pair of conflicting operations $op_i^1$ and $op_j^2$, $i \neq j$, and a sequence of operations $\tau$, if $op_i^1$ and $op_j^2$ are not properly ordered in $\tau$, then there exists a generalized data race between $op_i^1$ and $op_j^2$ in $\tau$.

We prove the main theorem of this section by way of contradiction, using the following lemma.

**Lemma 4.3** If $\tau$ is not fully admissible, then there exists a prefix of a sequential execution of $\text{Prog}$ which contains two operations that are not properly ordered.

**Proof:** Assume that $\tau$ is not fully admissible. We first prove the following claim, which locates the pair of operations that is a candidate for not being properly ordered.

**Claim 4.4** There exist two operations $op_i^2(x,v)$ and $op_k^1(x,w)$ in $\tau$ such that

1. $op_i^2(x,v) \rightarrow op_k^1(x,w)$,
4.3 Running GDRF Programs on Alpha Consistent Memories

Theorem 4.1 Every alpha consistent run of a GDRF program is sequentially consistent.

Proof: The proof of this theorem borrows ideas from the proof that every hybrid consistent run of a data race free program in sequentially consistent [9]. To prove this result, we consider a legal sequence of memory operations \( \tau \), as guaranteed in the definition of alpha consistency, that is minimal with respect to the number of switched operations (operations by the same processor \( p_i \) whose order in \( \tau \) is not consistent with \( p_i \)'s flow control sequence). We show that if \( \tau \) is not fully admissible (i.e., a sequential execution) then there exists a prefix of a sequential execution of the program that contains a generalized data race. If \( \tau \) is not fully admissible, it must contain at least one pair of switched operations. We locate the “first” pair of switched operations in \( \tau \), such that no other pair of switched operations is ordered between them. Because \( \tau \) is minimal we know this pair was switched to preserve legality. This fact is used to show that there is a generalized data race between some pair of operations that precedes this switched pair. Our main problem is to place these two operations (and the generalized data race between them) in a legal and partially admissible sequence. This is done by taking the two operations and the operations that influence them and ordering them as in \( \tau \), then adding any operations necessary to preserve the flow control sequences of all processors. The key point to prove about the resulting sequence is that it is indeed a prefix of a possible sequential execution. We do so by inductively fixing the objects that are accessed and the values written, so the register dependence is preserved; in addition, we fix the values returned by reads in order to make them legal. If there is a situation where this cannot be done, we show that we have a generalized data race earlier in the sequence. In this way, we construct a prefix of a sequential execution containing a generalized data race, which is a contradiction. The details of the proof follow.

In the proof, we need the following formalization of the notion of one operation influencing another, which relies on the flow control sequence.

Let \( \tau \) be a sequence of memory operations and let \( fcs_i \) be a flow control sequence that is consistent with \( \tau \), for each \( p_i \); let \( co_i \) be the derived control order. An operation \( op^1 \) directly influences an operation \( op^2 \) in \( \tau \) if one of the following holds:

1. \( op^1 \xrightarrow{cs_i} op^2 \) and \( op^1 \) is a read. (Note that \( j = k \) in this case.) That is, \( op^1 \) is a read operation which could affect the execution of \( op^2 \) through a control operation.

2. \( op^2 = r_k(y, v), op^1 = w_j(y, v), op^1 \xrightarrow{\tau} op^2 \) and there does not exist \( w_h(y, u) \) such that \( u \neq v \) and \( w_j(y, v) \xrightarrow{\tau} w_h(y, u) \xrightarrow{\tau} r_k(y, v) \). That is, \( op^2 \) is a read of the value written by \( op^1 \) and there is no intervening write of a different value.

3. \( op^1 \xrightarrow{fcs_i} op^2 \) and regs(\( op^1 \)) \cap regs(\( op^2 \)) \neq \emptyset. \) (Note that \( j = k \) in this case.) That is, the instructions that correspond to \( op^1 \) and \( op^2 \) touch the same local register.
- For two operations \( op_i^1 \) and \( op_j^2 \) in \( \tau \), denote \( op_i^1 \stackrel{P}{\rightarrow} op_j^2 \) if \( i = j \) and \( op_i^1 \stackrel{S}{\rightarrow} op_j^2 \) (this is the processor order).

- For a release sequence \( rel_i^1 \) and an acquire sequence \( acq_j^2 \) in \( \tau \), denote \( rel_i^1 \stackrel{\text{gb}}{\rightarrow} acq_j^2 \) if \( rel_i^1 \stackrel{r}{\rightarrow} acq_j^2 \) (this is the generalized synchronization order).

The generalized happens before relation, denoted by \( \stackrel{\text{gb}}{\rightarrow} \), is the transitive closure of the union of \( \stackrel{P}{\rightarrow} \) and \( \stackrel{S}{\rightarrow} \). This definition is similar to the definition of happens before in [3] and is closely related to the definition of happened before defined by Lamport [23] for message passing systems.

Two memory accesses conflict if they both access the same memory location and at least one of them is a write. A generalized data race occurs in a sequence of memory operations when two conflicting memory accesses that are not part of a release sequence or an acquire sequence are not ordered by the generalized happens before relation.

**Definition 4.1** A program is generalized data race free (GDRF) if none of its sequential executions contains a generalized data race.

### 4.2 MBV and MBP Sequences

We present an example of release and acquire sequences, MBV and MBP, which implement Dijkstra’s V and P operations on semaphores [14]. Specifically, some of the memory objects are semaphore objects. These objects may be accessed only by operations belonging to MBV and MBP sequences and each such sequence may access only one semaphore object. The sequential specification of a semaphore object includes all possible interleaving of MBV and MBP sequences such that in every prefix of the interleaving, the number of completed MBP sequences is larger than the number of completed MBV sequences by at most one.

Assuming that \( obj \) is a semaphore object initiated to 1 and \( reg \) is a local register, the following are possible MBV and MBP sequences:

<table>
<thead>
<tr>
<th>Code for MBV(obj)</th>
<th>Code for MBP(obj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB; vloop: ( reg = \text{ll}(obj) ); ( reg = reg + 1 ); ( \text{sc}(obj, reg) ); if the store was unsuccessful then goto vloop;</td>
<td>ploop: ( reg = \text{ll}(obj) ); if ( reg = 0 ) then goto loop; ( reg = reg - 1 ); ( \text{sc}(obj, reg) ); if the store was unsuccessful then goto ploop;</td>
</tr>
</tbody>
</table>

(We remind the reader that we use the shorthand notation \( \text{ll} \) for load-locked and \( \text{sc} \) for store-conditional.)
4 Running Generalized Data Race Free Programs

The notion of data race free programs is usually defined for memory models where the memory operations can be classified into synchronization (or strong) operations and data (or weak) operations [2, 3, 4, 19, 20, 26, 27]. Data operations are similar to reads and writes in alpha consistency. Synchronization operations, on the other hand, enforce a global ordering among themselves and on the ordering of data operations with respect to synchronization operations by the same processor. A sequence of operations is then said to be data race free if all “conflicting” data operations are ordered by synchronization operations. A program is data race free if none of its sequential executions contain a data race. This definition, however, is not applicable for alpha consistency, as alpha consistency does not include an obvious parallel for synchronization operations. Thus, we need to define sequences of operations that will imitate the effect of synchronization operations and adjust the definitions accordingly.

4.1 Generalized Data Race Free Programs

Let $R$ be an an alpha consistent run and let $\tau$ be a legal sequence of operations $\tau$ as guaranteed by the definition of alpha consistency (Definition 3.1). For some process $p_j$, consider a finite consecutive subsequence $\alpha$ of $\tau | j$. The subsequence $\alpha$ is a release sequence if the following hold:

1. the first operation in $\alpha$ is an MB operation,
2. all operations in $\alpha$ that are not MB operations access the same memory object, and
3. the last operation that is not an MB operation in $\alpha$ is either a successful write or a read.

Symmetrically, $\alpha$ is an acquire sequence if the following hold:

1. the last operation in $\alpha$ is an MB operation,
2. all operations in $\alpha$ that are not MB operations access the same memory object, and
3. the last operation that is not an MB operation in $\alpha$ is either a successful write or a read.

An example of such sequences appears below in Section 4.2.

Denote by $rel_i$ a release sequence by processor $p_i$ and by $acq_i$ an acquire sequence by processor $p_i$. That is, $rel_i$ includes all operations in the release sequence in the order they appear in the sequence; $acq_i$ includes all operations in the acquire sequence in the order they appear in the sequence. We slightly abuse the notation and denote by $op_i^j \xrightarrow{\tau} rel_k$ the fact that $op_i^j$ appears in $\tau$ before every operation in $rel_k$ and by $acq_k \xrightarrow{\tau} op_i^j$ the fact that $op_i^j$ appears in $\tau$ after every operation in $acq_k$. Similarly, $rel_k \xrightarrow{\tau} acq_j$ denotes the fact that all operations in $rel_k$ appear in $\tau$ before all operations in $acq_j$.

Let $\tau$ be a sequence of operations.
3.4 Control Order

The requirement that the sequence $\tau$ is consistent with the control order of each process (the first requirement in the definition of alpha consistency), is not an explicit part of the definition of the Alpha shared memory model that appears in [1, 32]. However, we believe that this requirement is an implicit part of the specification. This is because branch predictions are resolved by Alpha chips before the following instructions reach their write-back stage [25]. In other words, an operation does not have a permanent (global) effect before a previous conditional branch has been resolved. Also, the conditions in branches are resolved by examining the contents of local registers only. Therefore, any implementation will satisfy this condition anyway.

To understand why this requirement is also desirable, consider the example program in Figure 4, assuming $x$ and $y$ are initially 0. Assume this program is being executed on an alpha consistent memory. If we omit the first condition, then the definition would allow the run $R$ in Figure 5. To show $R$ is alpha consistent we take

$$\tau = w_2^2(x, 5), w_1^2(y, 5), r_1^1(x, 5), r_2^1(y, 5).$$

In this run, the values returned by the reads affect the decision whether to invoke the writes and the invocations of the writes affect the possible values returned by the reads. If such behavior is allowed, then writing programs and arguing about them is much more complicated. Hence, it is important to eliminate such runs, with circular inference relations where the prediction about the result of a control operation could affect its actual result.

In Section 4, we define GDRF programs and show that every alpha consistent run of a GDRF program is sequentially consistent (Theorem 4.1). According to these definitions, the program in Figure 4 is GDRF. If the first condition is omitted from the definition of alpha consistency, then the run $R$ in Figure 5 is alpha consistent. However, this run is not sequentially consistent. Hence, Theorem 4.1 is not valid if we omit the the requirement that the sequence $\tau$ is consistent with the control order.
the definition of alpha consistency, \( w_i(x, 2) \xrightarrow{\tau} w_j(x, 3) \xrightarrow{\tau} r_i(x, 3) \). Similarly, \( w_j(x, 3) \xrightarrow{\tau} r_k(x, 3) \xrightarrow{\tau} r_k(x, 2) \). A contradiction to the assumption that \( \tau \) is legal.

**Litmus test 4 (sequence okay):** According to litmus test 4, assuming that the initial value of both \( x \) and \( y \) is 1, the run \( R \) described in Figure 2 should be alpha consistent.

It is easy to verify that the following sequence of operations \( \tau \) obeys all requirements in the definition of alpha consistency (Definition 3.1):

\[
\tau = r_j(x, 1), w_i(x, 2), w_i(y, 2), r_j(y, 2)
\]

(We assume that \( \text{regs}(r_j(y, 2)) \cap \text{regs}(r_j(x, 1)) = \emptyset \).) Thus, \( R \) is an alpha consistent run.

**Litmus test 7 (impossible sequence):** According to litmus test 7, assuming that the initial value of both \( x \) and \( y \) is 1, the run \( R \) described in Figure 3 should not be alpha consistent.

We need to show that there does not exist a legal sequence of operations \( \tau \) that obeys all requirements in the definition of alpha consistency (Definition 3.1). Assume, by way of contradiction, that such a sequence exists. By the legality of \( \tau \) and the second requirement in the definition of alpha consistency,

\[
\begin{align*}
  w_i(x, 2) &\xrightarrow{\tau} MB_i & w_j(y, 1) &\xrightarrow{\tau} r_i(x, 1).
\end{align*}
\]

A contradiction to the assumption that \( \tau \) is legal.
3. If \( \text{op}_j \xrightarrow{f_{cs}} \text{op}_i \) and they both access the same object, then \( \text{op}_j \xrightarrow{\tau} \text{op}_i \).

4. If \( \text{op}_j \xrightarrow{f_{cs}} \text{op}_i \) and \( \text{regs}(\text{op}_j) \cap \text{regs}(\text{op}_i) \neq \emptyset \), then \( \text{op}_j \xrightarrow{\tau} \text{op}_i \).

**Remark:** The definitions in [9] include a choice of a set of memory operations, to support speculative evaluation of the code. Since Alpha decides on branches before the following operations reach their write-back stage, it does not rely on speculative execution. While this feature could have been easily added to the above definition we decided not to include it for the sake of transparency.

### 3.3 Relation to the Specification of Alpha

The Alpha Architecture Handbook ([1]) defines a processor issue order and a location access order for each processor and every memory location, respectively. These partial orders are defined with respect to “a hypothetical simple implementation that contains one processor and a single shared memory, with no caches or buffers” [1, page 5-10]. For each processor, there exists a partial order, called processor issue order, on all operations of this processor; every pair of operations by the same processor must be ordered by the processor issue order if at least one of them is an MB operation or they both access the same memory location. For each memory location, there exists a partial order, called location access order, on all operations to the same location; every pair of memory operations that are issued by the same processor to the same location are required to appear in the location access order in the same order as in the processor issue order. The transitive closure of these two partial orders is used to define the notions of before and after in the natural way.

Note that the notion of a flow control sequence in Definition 3.1 formally captures the order in which operations are issued by “a hypothetical simple implementation”. It can be seen that given a legal sequence of operations \( \tau \) as guaranteed in Definition 3.1 and two operations \( \text{op}_j \) and \( \text{op}_k \), if \( \text{op}_j \) is “before” \( \text{op}_k \) (in the sense defined above), then \( \text{op}_j \xrightarrow{\tau} \text{op}_k \). Similarly, if \( \text{op}_j \) is “after” \( \text{op}_k \), then \( \text{op}_k \xrightarrow{\tau} \text{op}_j \).

The Alpha Architecture Handbook describes nine litmus tests for verifying if a given run obeys the alpha shared memory model or not [1, pages 5-13]. As further evidence, we show that litmus tests 3, 4 and 7 are satisfied by Definition 3.1. The other litmus tests can be similarly verified.

**Litmus test 3 (impossible sequence):** According to litmus test 3, assuming that the initial value of \( x \) is 1, the run \( R \) described in Figure 1 should not be alpha consistent.

We need to show that there does not exist a legal sequence of operations \( \tau \) that obeys all requirements in the definition of alpha consistency (Definition 3.1). Assume, by way of contradiction, that such a sequence exists. By the legality of \( \tau \) and the third requirement in
a store or a store-conditional. A read operation by process \( p_j \) that reads a value \( v \) from object \( x \) is denoted \( r_j(x,v) \). Similarly, a write operation by process \( p_j \) that writes a value \( v \) to object \( x \) is denoted \( w_j(x,v) \).

Whenever we wish to emphasize that a read by \( p_j \) is a load operation that reads a value \( v \) from an object \( x \), we denote it by \( l_j(x,v) \). A load-locked operation by \( p_j \) that reads a value \( v \) from an object \( x \) is denoted by \( ll_j(x,v) \). A store operation by \( p_j \) that writes a value \( v \) to an object \( x \) is denoted by \( s_j(x,v) \). A store-conditional operation by \( p_j \) that attempts to write a value \( v \) to an object \( x \) is denoted by \( sc_j(x,v) \).

The memory of an Alpha based multiprocessor consists of a collection of \textit{alpha objects} and one special \textit{null} object. Given a sequence of operations \( \tau \), a store-conditional is labeled as either successful or unsuccessful, according to the following rule: A store-conditional \( sc_j(x,v) \) must be labeled as unsuccessful in \( \tau \) if one of the following holds:

a. there is no load-locked \( ll_j(x,u) \) such that \( ll_j(x,u) \xrightarrow{\tau} sc_j(x,v) \) or

b. there is another store or successful store-conditional \( w_k(x,w) \) such that \( w_k(x,w) \xrightarrow{\tau} sc_j(x,v) \) and there is no load-locked \( ll_j(x,u) \) ordered between them in \( \tau \).

Other store-conditional operations in \( \tau \) may be labeled as either successful or unsuccessful.\(^1\) A write is successful if it is either a store or a successful store-conditional; otherwise it is unsuccessful. The sequential specification of an alpha object includes every sequence \( \tau \) of read and write operations in which every read returns the value of the last successful write in \( \tau \), or the initial value of the object if there is no such write. A read is legal in a sequence of operations \( \tau \) if it returns the value of the last successful write to the same object in \( \tau \); otherwise, the read is illegal in \( \tau \).

A \textit{memory-barrier (MB)} operation always access the \textit{null} object. The sequential specification of the null object includes every possible interleaving of memory-barrier operations.

To define alpha consistency, we use the following notation: if \( \tau \) is a sequence of operations, and \( op_i \) precedes \( op_j \) in \( \tau \), we write \( op_i \xrightarrow{\tau} op_j \).

\textbf{Definition 3.1 (alpha consistency)} A run \( R \) is alpha consistent if there exists a set of flow control sequences \( \{fcs_j\}_{j=1}^m \) and a legal permutation \( \tau \) of all operations in \( R \) such that the following hold for any \( j \):

1. If \( op_j \xrightarrow{\tau} op_j \), then \( op_j \xrightarrow{\tau} op_j \).

2. If \( op_j \xrightarrow{fcs_j} op_j \) and either \( op_j \) or \( op_j \) is an MB operation, then \( op_j \xrightarrow{\tau} op_j \).

\(^1\)In order to capture the implementation considerations discussed above, this definition allows a store-conditional to be unsuccessful with no apparent reason.
3.1 Informal Description

The Alpha has four types of memory instructions: load, load-locked, store, and store-conditional. Intuitively, a load-locked operation reads from a memory location and marks this location. A store-conditional tries to store data into a memory location. If the memory location is still marked, then the memory location is updated and we say that the store-conditional is successful. Otherwise, the store-conditional has no affect on the memory location and we say that the store-conditional is unsuccessful. A store or a successful store-conditional always cancel the marking of the memory location it accesses (if such marking exists). This includes a store or a store-conditional by the processor that invoked the load-locked that caused the memory location to be marked.

Load-locked and store-conditional allow to implement sequences that imitate “atomic” read-modify-write operations in the following optimistic manner: First, the memory location is read into a local register using a load-locked. The result is then modified in the local register. Finally, the contents of the local register are written into the memory location using a store-conditional. If the store-conditional is successful, then it means that no other processor tried to store to the memory location between the load-locked and the store-conditional. Hence, the sequence has an “atomic” effect and it terminates. Otherwise, it starts all over again.

Note that in practice, a load-locked marks a complete cache line, which usually includes more than one memory location. Similarly, a store/store-conditional cancels the marking of a whole cache line. Thus, sometimes a store-conditional does not succeed with no apparent reason. This is because of another store/store-conditional writing into a memory location that is included in the same cache line and clearing the marking of the whole line.

Alpha is a highly optimized architecture which allows operations to be invoked in a pipeline and out of order. Every processor must “view” all operations in some order. However, the order in which operations of the same processor are viewed may be different from the order in which they were invoked (or the order they appear in the program). Whenever the order between two or more memory operations must be preserved, a memory-barrier (MB) operation must be used. It is guaranteed that all operations invoked by a processor before a memory-barrier operation will be viewed by all processors before the memory-barrier operation. Symmetrically, all operations invoked by a processor after a memory-barrier operation will be viewed by all processors after the memory-barrier operation. (This and the following requirement form what is called the program issue order in [1]).

Finally, two operations that are invoked by the same processor and access the same memory location must be “viewed” by all processors in the order implied for these operations by the program. (This is the location access order defined in [1].)

3.2 Alpha Consistency

We start with some simple notation. The operations allowed on an alpha object are load, store, load-locked, and store-conditional. A read is either a load or a load-locked and a write is either
immediately after $I$ in the program. A flow control sequence can either be finite or infinite (for non-terminating programs). Note that the flow control sequence implies a total order on the operations appearing in it; we denote this order by $\rightarrow_{fcs_i}$.

Let $fcs_i$ be a flow control sequence for $p_i$. A sequence $\tau$ of memory operations is fully $fcs_i$-admissible if $\tau|i$, the subsequence of $\tau$ consisting exactly of operations involving $p_i$, is equal to the subsequence of $fcs_i$ consisting exactly of the memory operations. Intuitively, this implies that the ordering of operations by $p_i$ in $\tau$ agrees with some flow control sequence $fcs_i$ for $p_i$, and does not end unless the program terminates. A sequence $\tau$ of memory operations is partially $fcs_i$-admissible if $\tau|i$ is a prefix of the subsequence of memory operations in $fcs_i$. Intuitively, this implies that, so far, the ordering implied by the flow control sequence is obeyed by $\tau$, but it is not necessarily completed yet.

A sequence $\tau$ of memory operations is fully (resp., partially) admissible with respect to a set of flow control sequences $\{fcs_i\}_{i=1}^n$, one for each $p_i$, if it is fully (resp., partially) $fcs_i$-admissible for all $i$.

A sequence of memory operations is a sequential execution if it is legal and fully admissible (with respect to some set of flow control sequences). The following is obvious:

Claim 2.1 Any legal partially admissible sequence of memory operations is a prefix of a sequential execution and vice versa.

For completeness, we include the following definition.

Definition 2.1 (sequential consistency) A run $R$ is sequentially consistent if there exists a sequential execution $\tau$ which is a permutation of all operations in $R$.

Based on the relation $fcs_i$, we define a partial order $\rightarrow_{cs}$ called the control order. Formally, for any two memory operations $op_i^1$ and $op_i^2$, $op_i^1 \rightarrow_{cs} op_i^2$ if there exists a control operation $op_i^3$ such that $op_i^1 \rightarrow_{fcs_i} op_i^3 \rightarrow_{fcs_i} op_i^2$.

3 The Alpha Shared Memory Model

We now present a formal definition of the shared memory model of an alpha based multiprocessor, following the description in [1, 32]. The definition ignores the issues of fetching instructions and updating the translation buffer and assumes that each object has an initial value.
Example: A read instruction, written as

\[ \text{read}(\text{op}_2), \]

consists of two operands: \( \text{op}_1 \) and \( \text{op}_2 \). \( \text{op}_1 \) is optional and must be a local register; the value returned by the read is stored into the local register (if it is given). \( \text{op}_2 \) is mandatory and may be either a local register or a constant value; the address to be accessed by this read is either the contents of \( \text{op}_2 \), if \( \text{op}_2 \) is a local register, or the value \( \text{op}_2 \), if \( \text{op}_2 \) is a constant value. A write instruction, written as

\[ \text{write}(\text{op}_1, \text{op}_2), \]

also consists of two operands: \( \text{op}_1 \) and \( \text{op}_2 \). Both \( \text{op}_1 \) and \( \text{op}_2 \) is mandatory and each one of them may be either a local register or a constant value; the address to be accessed by this write is either the contents of \( \text{op}_2 \), if \( \text{op}_2 \) is a local register, or the value \( \text{op}_2 \), if \( \text{op}_2 \) is a constant value; the value to be written by this write is either the contents of \( \text{op}_2 \), if \( \text{op}_2 \) is a local register, or the value \( \text{op}_2 \), if \( \text{op}_2 \) is a constant value.

Note that CISC architectures allow indirect addressing modes in which the operand is a memory address that holds the actual address to be accessed. The above definition does not allow these addressing modes; this does not impose a problem since Alpha is a RISC architecture.

An event is a call, a response, or a control operation (condition evaluation). A run (of the system) is a sequence of events such that there is a correspondence between calls and responses (matching object and process) and each response follows its corresponding call. Although a run is a sequence of events, it can also be viewed as containing operations. Each control operation is itself an event. The memory operations in a run are obtained by matching up corresponding call and response events; we assume that the run-time environment matches the call and response events defining the memory operations.

We sometimes use a shorthand notation for operations by different processes: An operation by process \( p_j \) is denoted by \( \text{op}_j \). When there is more than one operation by the same process, we use superscripts to distinguish among them, e.g., \( \text{op}_1^1, \text{op}_1^2, \) etc.

A sequence \( \tau \) of operations is legal if for each object \( x \), \( \tau|_x \), the subsequence of \( \tau \) consisting of exactly the operations involving \( x \), is in the sequential specification of \( x \). This captures the notion of satisfying the sequential specifications of the objects.

We next define the notion of being consistent with a sequential execution of the program. We build up inductively a run of \( p_i \)'s program in which every instruction finishes executing before the next one begins. Given processor \( p_i \)'s program, a flow control sequence, \( fcs_i \), is a sequence of operations defined as follows. The first element of \( fcs_i \) is an instance of the first instruction in \( p_i \)'s program. Suppose the \( k \)-th element of \( fcs_i \), denoted \( \text{op}_k \), is an instance of instruction \( I \) in the program. If \( \text{op}_k \) is a control operation and its condition evaluates to true, then the \( (k+1) \)-st element of \( fcs_i \) is an instance of the instruction whose label is the branch of instruction \( I \). Otherwise the \( (k+1) \)-st element of \( fcs_i \) is an instance of the instruction
Attiya, Chaudhuri, Friedman and Welch showed how to transform any non-cooperative mutual exclusion algorithm based on sequential consistency into an algorithm based on hybrid consistency [9]. Their method yields slightly more efficient algorithms than the general method developed here. However, hybrid consistency is a theoretical consistency condition while alpha consistency describes the shared memory model of an existing commercial architecture.

2 Framework

An application program consists of a sequence of instructions, each with a unique label. There are two types of instructions, (shared) memory instructions and control instructions. A memory instruction specifies an access to a shared object. The specific kind of access depends on the data type of the object. A control instruction consists of a condition (a boolean function of the processor’s local state) and a branch (jump to the instruction with the given label).

The memory consistency system (MCS) implements the shared objects that are manipulated by the application programs. It consists of a processor at each node as well as possibly other hardware. The interface to the MCS consists of calls (also called invocations) and responses on particular objects.

Every object is assumed to have a sequential specification (cf. [21]) defining a set of (memory) operations, which are ordered pairs of calls and responses, and a set of (memory) operation sequences, which are the allowable sequences of operations on that object.

The run-time environment at a node takes as input an application program and executes instructions on the MCS. An operation is a specific instance of an execution of an instruction. A memory operation consists of two parts, a call (to the MCS process) and a matching response (from the MCS process). A control operation consists of an evaluation of its condition. A control operation is represented by the result (true or false) of the evaluation. Thus the run-time environment must keep track of the local state of the application process in order to perform the evaluation.

Every node maintains a (possibly infinite) set of local registers, accessible to this node only. We do not model the local registers explicitly, but assume that they may be manipulated by memory and control instructions. In particular, values may be assigned to local registers and be retrieved from them. The contents of local registers may be used as addresses for memory objects, as values to be assigned to memory objects, or to temporarily store values read from memory objects. We assume that each memory instruction may include several operands, each of which is either a local register or a constant value. In case one of the operands of a memory instruction is a local register, we say that the memory instruction touches the local register.

Given a memory operation $op$, we denote by $\text{regs}(op)$ the set of local registers touched by the corresponding memory instruction.
uncertainty of concurrent programs. There has been a wide study of methods to detect and report data races [6, 11, 16, 15, 24, 26, 27]. It is reasonable to assume that generalized data race free programs account for a substantial portion of all concurrent programs.

Another useful concurrent programming technique is to protect accesses to shared data inside critical sections. A critical section is a piece of code that only one processor may execute at any time. Guaranteeing mutual exclusion for critical sections is a well known problem that has been investigated quite thoroughly, e.g., [13, 22, 29, 30]. Our second rule is a general method for transforming any non-cooperative mutual exclusion algorithm based on sequential consistency into a mutual exclusion algorithm based on alpha consistency. This method is based on inserting memory-barrier operations in several places in the code.

Although this rule is very general, it is not optimal in terms of the number of memory-barrier operations. We examine two well-known mutual exclusion algorithms: Peterson’s algorithm [29] and Lamport’s bakery algorithm [22]. In both cases, a specific analysis of the code requires less memory-barrier operations than inserted by the general method.

1.3 Related Work

Our work joins two previous works in attempting to provide a formal memory model for an existing architecture and using this model to develop programming techniques for systems based on this architecture. The first is for the Stanford DASH multiprocessor [19, 20] and the second is for PowerPC based multiprocessors [12].

Gharachorloo, Gibbons and Merritt define release consistency, a formal definition of the memory model of the Stanford DASH multiprocessor. They show that PL-programs run on release consistency as if it was sequentially consistent [19, 20]. PL-programs are similar to generalized data race free programs. There is a slight difference between the way release and acquire operations and the interaction between them are defined in alpha consistency and in release consistency. In alpha consistency, release and acquire are sequences of operations while in release consistency, release and acquire are assumed to be indivisible atomic operations. In addition, PL-programs require a pairing to be defined between release and acquire operations; alpha consistency does not impose a similar requirement.

Corella, Stone and Barton provide an axiomatic definition of the memory model of PowerPC based multiprocessors [12]. They show that if all accesses to shared data on a PowerPC based multiprocessor are protected with lock/unlock sequences, then the result is as if the hardware was sequentially consistent. Protecting all access with lock/unlock sequences is a special case of generalized data race free programs; a lock sequence is the same as an MBP sequence and an unlock sequence in the same as an MBV sequence.

The idea of showing that data race free programs run on an optimized hardware as if the hardware was sequentially consistent was pioneered by Advé and Hill [3, 5]. It has also been investigated in the context of several other shared memory models [7, 9, 19, 20, 31].
1 Introduction

1.1 Overview

Alpha is a novel 64-bit load/store RISC architecture, introduced by Digital Equipment Corporation in 1992. Alpha was designed with particular emphasis on multiple instruction issue and support for multi-processing. It is one of the first commercial microprocessors planned to serve as a building block for multiprocessors. Therefore, its architectural specification includes a detailed description of the shared memory behavior of multiprocessors based on Alpha chips. Although this specification is rigorous, it requires an understanding of many parts of the architecture and employs notions of ordering accesses, etc. As such, it is very comfortable for hardware designers, but is not very convenient for programmers wishing to exploit the improved efficiency of Alpha.

To make Alpha more accessible to programmers, we present a formal and crisp definition of its shared memory model. We also present several programming techniques that allow to run programs written assuming sequentially consistent hardware on alpha consistent hardware.

1.2 Detailed Description

Our formal definition of the shared memory model of Alpha based multiprocessors is called alpha consistency. Alpha consistency is defined in the framework of [9] and allows arbitrary out-of-order execution of instructions. The memory supports simple load and store operations, sophisticated load-locked and store-conditional operations, as well as memory-barrier operations which allow efficient synchronization. Our definition describes in a simple manner the order in which operations are observed by different processors. Therefore, we believe it is more convenient for programmers than the specification appearing in the Alpha architecture handbook [1].

Our definition attempts to capture the semantics intended in [1], with one important exception: it ignores the issues of fetching instructions and updating the translation buffer. It is not clear how the inclusion of these issues will affect our results.

The formal definition of alpha consistency allows us to develop general techniques for programming Alpha based multiprocessors. We show several rules which allow to use a program designed for sequentially consistent machines on an alpha consistent machine. Such rules release the programmer from the burden of considering the specific intricacies of alpha consistency.

The first rule is to run data race free programs. We show that generalized data race free programs run on alpha consistent hardware as if it was sequentially consistent. Data races are avoided by inserting synchronization sequences; a specific example of synchronization sequences is MBP and MBV sequences, which generalize Dijkstra’s P and V operations on semaphores. Programs that exhibit data races are considered bad practice since they add to the inherent
Abstract

Alpha consistency, a formal definition of the shared memory model of DEC-Alpha based multiprocessors, is defined in the framework of [9]. The definition allows arbitrary out-of-order execution of instructions and is convenient for use by programmers. Several programming techniques that allow to run programs written assuming sequentially consistent hardware on alpha consistent hardware are presented. First, it is shown that every data race free program runs on an alpha consistent memory as if it was sequentially consistent. Second, it is shown how to transform any non-cooperative solution for the mutual exclusion problem based on sequential consistency into a solution based on alpha consistency. Finally, it is shown that this general method is not optimal; for two well-known mutual exclusion algorithms, a specific analysis of the code yields a more efficient algorithm.
Programming DEC-Alpha Based Multiprocessors The Easy Way

Hagit Attiya  Roy Friedman

Department of Computer Science
The Technion
Haifa 32000, Israel

October 16, 1994

1This research was partially supported by grant No. 92-0233 from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel, Technion V.P.R.—Argentinian Research Fund and the fund for the promotion of research in the Technion.

2Author's current affiliation: Department of Computer Science, Cornell University, Ithaca, NY 14853.