


References


In this paper we have tried to bridge over these two contradictory goals. We presented a general framework which encompasses the functionality of the compiler and the run-time environment and models their interaction with the memory consistency system. Our framework allows the definition of known consistency conditions to be combined with implementations that exploit optimizations for reducing the latency of memory accesses. To the best of our knowledge, our definitions are unique in modeling the whole program, rather than just looking at the memory operations in isolation.

We also characterized requirements on programs that guarantee that they will behave on hybrid consistent memories as if they are sequentially consistent. This allows programmers to reason about certain classes of programs assuming sequential consistency, yet run them on more efficient hardware.

This work is part of an on-going attempt to understand consistency conditions and their implications on programming, compiler design and architecture. Much research is still needed before this goal can be met. While more efficient, fault-tolerant algorithms for implementing various consistency conditions still need to be developed, our paper takes a complementary approach: it provides a clean and formal framework for investigating systematic methods, rules and compiler techniques to transform programs written for strong consistency conditions into correct programs for weaker consistency conditions.

Acknowledgements: We would like to thank Kourosh Gharachorloo for helpful comments on a previous version of this paper.
The second drawback is that the consistency conditions are then bound to the optimizations that are specifically mentioned by the definition. In our framework, the hardware optimizations are not part of the consistency condition, contributing to programs that are more portable and whose correctness is independent of advances in technology.

The work of Adve and Hill deals formally with non-sequential execution of memory operations, although not explicitly. In particular, while the sufficient condition for DRF1 \cite{3, 4} includes a formal treatment of control, this condition is based on the notion of a read operation controlling a write operation by the same processor. This is an operational notion and it is not proven that it captures all the possible ways one operation can control another. In contrast, our approach is syntactic, based on using the actual control instructions in the program, and thus is safer. Most previous work on specifying consistency conditions has either totally ignored control instructions \cite{6, 9, 10, 27, 25} or has merely made the intuitive informal requirement that uniprocessor control dependences are preserved \cite{2, 19, 20, 21}.

The work of Gibbons and Merritt \cite{20} deals formally and explicitly with pipelining of memory operations. In their definition, the program is not explicitly modeled, and it is not clear how the intended semantics of the program is preserved. In contrast, our framework explicitly models the program and the run-time system executing it. Unlike our paper, their results do not encompass arbitrary out-of-order or speculative execution of operations. Yet recent experiments (e.g., \cite{26, 41}) have shown that without speculative execution, significant speedup due to parallelism cannot be achieved for programs with complex control flow. Thus it is important to allow for it.

The approach of programming with hybrid consistency by running data-race-free programs was pioneered by Adve and Hill \cite{2}, where they focused on implementations. We have applied this approach to the programmer’s interface. Similar theorems have been proved in other papers \cite{2, 3, 21, 20}, showing that different types of data-race-free programs can be executed on more relaxed implementations of shared memory as if they were sequentially consistent. In our opinion, our result is the only one that combines full support for non-sequential execution, a more natural interface (namely that between the program and the mcs), with being short and comprehensible while still rigorous. We are not aware of any previous technique that, like our static approach, allows labeling of operations without considering all possible executions of the program.

8 Discussion

As the demand for powerful computers grows faster than the technology to develop new processors, the need for highly parallel multiprocessors increases. However, in order to fully utilize such machines, convenient paradigms for writing concurrent programs must be developed. These paradigms should allow the user to enjoy the same simplistic model of the world as in uniprocessors, without sacrificing the performance of the whole system. These two goals are somewhat contradictory. Recent results indicate that there is a tradeoff between the similarity of a distributed shared memory to real shared memory, and the efficiency of the hardware.

\footnote{We eschew the term non-blocking used by Gibbons and Merritt since it conflicts with an earlier usage of this term with a completely different meaning by Skeen \cite{38} and Herlihy \cite{24}.}
The proof of the following theorem follows from Lemma 6.1 exactly like the proof of Theorem 5.5.

**Theorem 6.5** Every asymmetric hybrid consistent run of an asymmetric data-race-free program is sequentially consistent.

7 Related Work

Many existing formal treatments of memory consistency conditions [25, 27, 9, 21] assume that memory operations are executed sequentially—one at a time and in program order. Several recent papers proposed formalisms to allow some non-sequential optimizations [2, 3, 19, 21, 20], and contain many similarities with our work. In this section we compare our work with these formalisms.

These formalisms may be partitioned into two categories: total consistency conditions and partial consistency conditions. Total consistency conditions define the behavior of their implementations for all programs, while partial consistency conditions merely require that certain programs will behave as if the hardware is sequentially consistent. Thus, they are not defined for programs that do not obey a specific condition, even if these programs are sensible. The first category includes release consistency [19, 21, 20], and the sufficient conditions for implementing DRF0 [2] and DRF1 [3]. The second category includes DRF0 [2], DRF1 [3], and PL-programs [21, 20].

Our framework provides total consistency conditions. In Section 4 we gave examples of programs that are neither DRF0, DRF1, PL-programs nor data-race-free programs. Yet, these programs are correct and efficient under hybrid consistency. Hence, it is interesting to have the consistency conditions be defined for any program.

The major feature distinguishing our approach is the focus on the specification of the conditions in a way that is machine-independent and high-level enough for the programmer. This is the reason our definitions are specified at the interface between the application program and the system. (This is along the lines of [27, 25, 9].) Other formalizations of total consistency conditions [2, 3, 4, 21, 20] focus on hardware implementations that can guarantee certain conditions efficiently.

Defining consistency conditions at the interface between the mcs and the interconnection network, as is done in the definition of weak ordering [15], the definition of release consistency [20, 21, 19], the sufficient conditions for DRF0 [2] and the sufficient conditions for DRF1 [3], has two severe drawbacks. First, this method leads to very detailed and complex definitions and makes it almost impossible to reason directly with the consistency conditions or to argue about the correctness of programs running on implementations of these conditions (see also in [17, page 2]); in contrast, our framework allows one to reason directly with the consistency conditions.

See for example the short and formal proof of correctness for a program that solves the mutual exclusion problem, for any hybrid consistent memory, in [9].
Now, we construct a new partially admissible (with respect to the $fcs_j$'s) sequence $\pi'$ by adding to $\pi$ every operation $op_l^i$ not in $\pi$, such that $op_l^i \xrightarrow{\text{fcs}_j} op_l^i$ and $op_l^i$ is in $\pi$, for each process $l$, as follows. For all such operations that were originally in $\tau'_i$, add it in the same place as in $\tau'_i$. For all other operations, add them arbitrarily, maintaining consistency with $\xrightarrow{\text{fcs}_j}$, for all $l$.

We would now like to determine if $\pi'$ is a legal sequence. By the second property of the influence relation and $\tau'_i$, $\pi$ is a legal sequence. In particular, all reads in $\pi$ are legal. Thus, if there are illegal reads in $\pi'$, then each illegal read is either a read that was added to $\pi$, or a read that became illegal because of some write that was added to $\pi$.

Consider every illegal read in $\pi'$ that was added to $\pi$, and change its value to match the value of the most recent write. Call this sequence $\pi''$. Let $r$ be one of these fixed-up reads. As in the proof of Lemma 5.1, we can show that the original reads do not influence any operation in $\pi''$, and hence $\pi''$ is partially admissible with respect to some set of flow control sequences $fcs_j$. So, if there are no illegal reads in $\pi''$, then $\pi''$ is legal and it follows that it is a prefix of a sequential execution of $\text{Prog}$.

To complete the proof of the lemma when $\pi''$ is legal, we now show that $\pi''$ contains an asymmetric data race between $op_l^2(x, v)$ and $op_l^1(x, w)$.

By Claim 6.2, there is no release operation by $p_j$ between $op_l^2$ and $op_l^1$ inclusive in $\tau'_i$. Also, $j \neq k$, thus $op_l^2$ and $op_l^1$ are not ordered by the happens before relation.

We are left with the case in which there is an illegal read in $\pi''$. It became illegal due to the insertion of some write. Let $r^1(z, v_1)$ be the first illegal read in $\pi''$, and let $w^1(z, v_2)$, $v_1 \neq v_2$, be the corresponding inserted write. Denote by $\sigma$ the shortest prefix of $\pi''$ which includes $r^1(z, v_1)$. Let $\sigma'$ be the same sequence where $r^1(z, v_1)$ is replaced by $r^2(z, v_2)$. We complete the proof by showing:

**Claim 6.4** $\sigma'$ is a prefix of a sequential execution of $\text{Prog}$ which contains an asymmetric data race between $r^2_q$ and $w^2_p$.

**Proof:** The proof is similar to the proof of Claim 5.4. We show that $r^1_q \xrightarrow{\tau_r} w^1_p$ by using exactly the same arguments as in the proof of Claim 5.4. Next, we need to show that $r^1_q(z, v_1)$ and $w^2_p(z, v_2)$ are not ordered by the happens before relation in $\pi''$, and hence, they are not ordered by the happens before relation in $\sigma$.

Suppose they were. As in the proof of Claim 5.4, we have that $p \neq q$. Thus, $w^1_p$ and $r^1_q$ must be related by $\xrightarrow{\text{fcs}_j}$. It follows that there is a release operation $rep_p$ and an acquire operation $aop_q$ between them in $\sigma$, i.e., $w^1_p \xrightarrow{\sigma} rep_p \xrightarrow{\sigma} aop_q \xrightarrow{\sigma} r^1_q$. By construction of $\sigma$ and definition of asymmetric hybrid consistency, $w^1_p \xrightarrow{\tau_l} rep_p \xrightarrow{\tau_l} aop_q \xrightarrow{\tau_l} r^1_q$. But this contradicts the fact that $r^1_q$ precedes $w^1_p$ in $\tau_l$.  

\[\blacksquare\]
6.3 Running Asymmetric Data-Race-Free Programs on Asymmetric Hybrid Consistent Memory

We prove that every asymmetric hybrid consistent run of an asymmetric data-race-free program is sequentially consistent. The proof follows the proof of Theorem 5.5 very closely.

Fix an asymmetric data-race-free program \textit{Prog}, an asymmetric hybrid consistent run \( R \) of \textit{Prog}, and as guaranteed by the definition of asymmetric hybrid consistency, a subset \( S \) of the memory operations and a flow control sequence \( fcs_i \) for each process \( p_i \).

Fix a particular process \( p_i \). Let \( T \) be the set of all operation sequences \( \tau_i \) that satisfy Definition 6.1 (asymmetric hybrid consistency). For \( \tau_i \) in \( T \), an ordered pair of operations of \( p_j \), \(< op_j^2, op_j^1 > \), is switched in \( \tau_i \) if \( op_j^2 \xrightarrow{\tau_i} op_j^1 \), but \( op_j^1 \xleftarrow{fcs_j} op_j^2 \).

Let \( \tau_i \) be some element of \( T \) with a minimal set of switches. That is, there does not exist \( \tau'_i \in T \), such that the set of pairs of switched operations of all processes in \( \tau'_i \) is strictly contained in the set of pairs of switched operations of all processes in \( \tau_i \). We prove the main theorem of this section by way of contradiction, using the following lemma.

**Lemma 6.1** If \( \tau_i \) is not fully admissible, then there exists a prefix of a sequential execution of \textit{Prog} which contains an asymmetric data race.

**Proof:** Assume that \( \tau_i \) is not fully admissible. We first prove the following claim, which locates the pair of operations that is a candidate for an asymmetric data race.

**Claim 6.2** There exist two operations \( op_j^2(x, v) \) and \( op_k^1(x, w) \) in \( \tau_i \) such that

1. \( op_j^2(x, v) \xrightarrow{\tau_i} op_k^1(x, w) \),
2. there is an asymmetric data race between \( op_j^2(x, v) \) and \( op_k^1(x, w) \), and
3. there is no pair of switched operations in \( \tau_i \) up to \( op_k^1(x, w) \).

**Sketch of proof:** The proof is similar to the proof of Claim 5.2; one should only note that in the definition of an asymmetric data race, similarly to the original definition of a data race, if two operations \( op_j^2 \) and \( op_k^1 \), \( j \neq k \), conflict, then there must be a release operation by \( p_j \) between them in order to avoid an asymmetric data race.

Let \( I \) be the set of operations in \( \tau_i \) that influence either \( op_j^2 \) or \( op_k^1 \) (including \( op_j^2 \) and \( op_k^1 \)). Let \( \tau'_i \) be the shortest prefix of \( \tau_i \) that includes every operation in \( I \). Let \( \pi \) be the subsequence of \( \tau'_i \) which contains exactly the set of all operations in \( I \). The following claim shows that \( \pi \) is consistent with \( fcs_i \), for all \( i \); its proof is exactly the same as the proof of Claim 5.3.

**Claim 6.3** For every pair of operations \( op_1^k \) and \( op_2^k \), if \( op_1^k \xrightarrow{\tau_i} op_2^k \) then \( op_1^k \xleftarrow{fcs_i} op_2^k \).
1. $\tau_i$ is fully $fcs_i$-admissible.

2. If $op_j^1 \xrightarrow{\tau_i} op_j^2$, then $op_j^1 \xrightarrow{\tau_i} op_j^2$, for any $j$.

3. If $op_j^1 \xrightarrow{fcs_j} op_j^2$, and either $op_j^1$ is an acquire or $op_j^2$ is a release, then $op_j^1 \xrightarrow{\tau_i} op_j^2$, for any $j$.

4. If $rop_j^1 \in S$ and $aop_k^2 \in S$ are paired, then $rop_j^1 \xrightarrow{\tau_i} aop_k^2$, for any $j$ and $k$.

5. If $op_j^1 \xrightarrow{fcs_j} op_j^2$ and $op_j^1$ and $op_j^2$ are memory operations accessing the same location, then $op_j^1 \xrightarrow{\tau_i} op_j^2$, for any $j$.

The only difference between this definition and the definition of hybrid consistency (Definition 3.3) is in the third and fourth properties. The third property distinguishes between the asymmetric effects of acquire and release operations, while the fourth property guarantees the ordering only for strong operations that are paired. As discussed in Section 5.3, this definition already includes the additional property requiring all processes to view all accesses of one process to the same location in the same order.

As it is defined now, asymmetric hybrid consistency is not a generalization of hybrid consistency. This is because the pairing relation is defined statically here and may not vary in different runs. So, while hybrid consistency allows a different total order of the strong operations in different runs, our definition of asymmetric hybrid consistency restricts the ordering of release and acquire operations to respect the same pairing relation in every run. We would like to modify the definition so asymmetric hybrid consistency would be a generalization of hybrid consistency. Doing so will probably require a more explicit treatment of the pairing relation, which is missing in the current literature.

### 6.2 Definition of Asymmetric Data-Race-Free Programs

Let $op_j^1$ and $op_k^2$ be two operations appearing in some sequence of memory operations $\alpha$. Then

- $op_j^1 \xrightarrow{\alpha} op_k^2$ if $j = k$ and $op_j^1 \xrightarrow{\alpha} op_k^2$.
- $op_j^1 \xrightarrow{\alpha*} op_k^2$ if both $op_j^1$ and $op_k^2$ are paired release and acquire and $op_j^1 \xrightarrow{\alpha} op_k^2$.

The relation *asymmetric happens before*, denoted by $\xrightarrow{bl^*}$, is the transitive closure of the union of $\xrightarrow{\beta^*}$ and $\xrightarrow{\alpha*}$. An *asymmetric data race* occurs when two conflicting memory accesses are not ordered by the relation $\xrightarrow{bl^*}$.

**Definition 6.2** A program is asymmetric data-race-free if none of its sequential executions contains an asymmetric data race.
Consider the following run $R$.

$$
R|_{p_1} : r_1^1(x, 5), w_1^2(x, 5), w_1^3(y, 5)
$$
$$
R|_{p_2} : r_2^1(y, 5), w_2^2(y, 5), w_2^3(x, 5)
$$

Although $R$ is not sequentially consistent (we leave it to the reader to verify this), it is hybrid consistent:

$$
\tau_1 = w_2^3(y, 5), r_2^1(y, 5), w_2^3(x, 5), r_1^1(x, 5), w_1^3(y, 5)
$$
$$
\tau_2 = w_1^3(x, 5), r_1^1(x, 5), w_1^3(y, 5), r_2^1(y, 5), w_2^3(y, 5), w_2^3(x, 5)
$$

6 Asymmetric Hybrid Consistency

In this section we present a (relatively minor) modification of the definition of hybrid consistency that distinguishes between release and acquire operations. We then modify the definition of a data race to accommodate this distinction. We conclude by proving that asymmetric data-race-free programs behave on asymmetric hybrid consistent memory implementations as if they were sequentially consistent. The proof is almost verbatim the same proof as for Theorem 5.5.

We believe our definition of asymmetric hybrid consistency captures a similar semantics to what is captured by the sufficient conditions presented by Adv and Hill [3]. We believe it also captures the main ingredients of the intended semantics of the Stanford Dash multiprocessor [19], as formalized in [21, 20]. Our definition, like the definition in [21, 20], does not include nsync operations.\footnote{More precisely, nsyncs are handled exactly like syncs (strong operations) in [21, 20], a solution that works because of the strong assumptions made in those papers, namely no pipelining of reads [21] and unique sequence numbers on values written [20]. As we pointed out in Section 4, these are significant assumptions.} See Section 7 for a discussion of similar results that have been proved by others.

6.1 Definition of Asymmetric Hybrid Consistency

We assume it is possible to mark some strong instructions in a program as releases and some as acquires (an operation can be both a release and an acquire, but this is not necessary). An instance of a release instruction is a release operation and an instance of an acquire instruction is an acquire operation. Furthermore, there is a pairing (a binary relation) defined between release operations and acquire operations.

We use the following additional notation. By $rop_p$ we denote a release operation invoked by process $p$, and by $aop_p$ we denote an acquire operation invoked by process $p$.

**Definition 6.1 (Asymmetric hybrid consistency)** A run $R$ is asymmetric hybrid consistent if there exists a subset $S$ of the memory operations in $R$, and a set of flow control sequences $\{fcs_i\}_{i=1}^n$, one for each $p_i$, such that for each process $p_i$, there exists a legal permutation $\tau_i$ of $S$ with the following properties:
contradiction. Therefore, $r^1_\phi(z, v_1)$ and $w^1_\phi(z, v_2)$ are not ordered by the happens before relation in $\pi''$, and hence, in $\sigma$, and there exists a data race between them in $\sigma$. It follows that $\sigma'$ contains a data race.

Now, $\sigma'$ is a legal sequence of operations due to the change in the value returned by the read, implied by replacing $r^1_\phi$ with $r^2_\phi$. Since $r^2_\phi(z, v_2)$ is the last operation in $\sigma'$, it clearly does not influence any operation in $\sigma'$. By Lemma 2.3, $\sigma'$ is partially admissible for some set of flow control sequences $\{fc_{\sigma_i}\}_{i=1}^n$. Therefore, $\sigma'$ is a prefix of sequential execution of $\text{Prog}$ which contains a data race between $r^2_\phi(z, v_2)$ and $w^1_\phi(z, v_2)$, as needed.

The proof of the following theorem is now immediate.

**Theorem 5.5** Every hybrid consistent run of a data-race-free program is sequentially consistent.

**Proof:** Let $R$ be a hybrid consistent run of a data-race-free program $\text{Prog}$. Fix a subset $S$ of the memory operations in $R$ and a set of flow control sequences $fc_{\sigma_i}$ from the definition of hybrid consistency. Choose a process $p_i$. Let $\tau_i$ be a minimal legal permutation of $S$ as guaranteed by the definition of hybrid consistency. Since $\text{Prog}$ is data-race-free, no sequential execution has a data race. Then by Lemma 5.1, $\tau_i$ is fully admissible. Thus, $R$ is a sequentially consistent run of $\text{Prog}$.

### 5.3 Reordering Process's Operations

The original definition of hybrid consistency in [9] did not include the last property (in addition to not considering control operations). Thus two weak operations by the same process $p_i$ accessing the same location could be viewed by other processes in a different order from the order in which they appear in $fc_{\sigma_i}$. We show that under this behavior it is not true that every hybrid consistent run of a data-race-free program is sequentially consistent.

Consider the following data-race-free program, assuming $x$ and $y$ are initially 0 and all instructions are weak:

<table>
<thead>
<tr>
<th>$p_1$'s program</th>
<th>$p_2$'s program</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{tmp}_1 := \text{read}(x)$;</td>
<td>$\text{tmp}_2 := \text{read}(y)$;</td>
</tr>
<tr>
<td>$\text{write}(x, 5)$;</td>
<td>$\text{write}(y, 5)$;</td>
</tr>
<tr>
<td>if $\text{tmp}_1 = 5$ then</td>
<td>if $\text{tmp}_2 = 5$ then</td>
</tr>
<tr>
<td>$\text{write}(y, 5)$;</td>
<td>$\text{write}(x, 5)$;</td>
</tr>
</tbody>
</table>
We would now like to determine if \( \pi' \) is a legal sequence. By the second property of the influence relation and \( \tau_i \), \( \pi \) is a legal sequence. In particular, all reads in \( \pi \) are legal. Thus, if there are illegal reads in \( \pi' \), then each illegal read is either a read that was added to \( \pi \), or a read that became illegal because of some write that was added to \( \pi \).

Consider every illegal read \( r_i \) in \( \pi' \) that was added to \( \pi \). We claim that \( r_i \) does not influence any operation in \( \pi' \), with respect to the \( co \)’s. Suppose it does. Since \( r_i \) is a read, it must influence an operation through a control relation, i.e., \( r_i \xrightarrow{co} op_l \), for some \( l \). Since \( r_i \) is not in \( I \), \( op_l \) is not in \( I \). Thus \( op_l \) was included in \( \pi' \) because there exists \( op_{l'} \in I \) such that \( op_l \xrightarrow{fcs} op_{l'} \). This implies that \( r_i \xrightarrow{co} op_{l'} \) and that \( r \) is in \( I \). We therefore have a contradiction.

Now, for each of the illegal reads in \( \pi' \) that were added to \( \pi \), change its value to match the value of the most recent write. Call this sequence \( \pi'' \). Apply Lemma 2.3 successively for each fixed-up read, starting with \( \pi'' \), to deduce that \( \pi'' \) is partially admissible with respect to some set of flow control sequences \( \{fcs'_i\}_i \). So, if there are no illegal reads in \( \pi'' \), then \( \pi'' \) is legal and it follows that it is a prefix of a sequential execution of \( \text{Prog} \).

To complete the proof of the lemma when \( \pi'' \) is legal, we now show that \( \pi'' \) contains a data race between \( op_j(z, v) \) and \( op_k(x, w) \). By Claim 5.2, there is no strong operation by \( p_j \) between \( op_j \) and \( op_k \) inclusive in \( \tau'_i \); also, note that no strong operation was added to \( \pi \). Also, \( j \neq k \), thus \( op_j \) and \( op_k \) are not ordered by the happens-before relation.

We are left with the case in which there is an illegal read in \( \pi'' \). It became illegal due to the insertion of some write. Let \( r_i^j(z, v_1) \) be the first illegal read in \( \pi'' \), and let \( w_i^j(z, v_2) \), \( v_1 \neq v_2 \), be the corresponding inserted write. Denote by \( \sigma \) the shortest prefix of \( \pi'' \) which includes \( r_i^j(z, v_1) \). Let \( \sigma' \) be the same sequence where \( r_i^j(z, v_1) \) is replaced by \( r_i^j(z, v_2) \). We complete the proof by showing:

**Claim 5.4** \( \sigma' \) is a prefix of a sequential execution of \( \text{Prog} \) which contains a data race between \( r_i^j \) and \( w_i^j \).

**Proof:** We first show that \( r_i^j \xrightarrow{\tau_i} w_i^j \). Assume otherwise that \( w_i^j \xrightarrow{\tau_i} r_i^j \). Since \( w_i^j \) is not in \( I \), it must not influence \( r_i^j \) in \( \tau_i \). Therefore, there is another write \( w_2(z, v_1) \) ordered between \( w_1^j \) and \( r_i^j \) which influences \( r_i^j \) in \( \tau_i \). But, then \( w_1^j \), \( w_2^j \) and \( r_i^j \) would be ordered similarly in \( \pi' \). This gives a contradiction to the assumption that \( r_i^j \) becomes illegal due to the insertion of \( w_i^j \).

So we can assume that \( r_i^j \xrightarrow{\tau_i} w_i^j \); it follows that \( w_i^j \) is not in \( \tau_i \), implying that \( w_i^j \) is weak. We claim that \( p \neq q \). Otherwise, if \( p = q \), then \( w_i^j(z, v_2) \xrightarrow{fs} r_i^j(z, v_1) \). By the last property of hybrid consistency, it follows that \( w_i^j(z, v_2) \xrightarrow{\tau_i} r_i^j(z, v_1) \), a contradiction. Therefore, \( p \neq q \).

We now show that \( r_i^j(z, v_1) \) and \( w_i^j(z, v_2) \) are not ordered by the happens before relation in \( \pi'' \), and hence, they are not ordered by the happens before relation in \( \sigma \). Suppose they were. Then, there is a strong operation \( sop_k \) between \( w_i^j \) and \( r_i^j \) in \( \pi'' \). Therefore, \( w_i^j \xrightarrow{fs} sop_k \) and thus \( w_i^j \xrightarrow{\tau_i} sop_k \). Therefore, \( sop_k \) is not in \( \tau_i \) since \( w_i^j \) is not in \( \tau_i \), and hence is weak, a
3. there is no pair of switched operations in \( \tau_i \) up to \( op_1^i(x, w) \).

**Proof:** Since \( \tau_i \) is not fully admissible, there exists at least one pair of switched operations in \( \tau_i \). Let \( \langle op_1^j(x, v), op_1^j(y, u) \rangle \) be the first ordered pair of switched operations, i.e., there is no other pair of switched operations which is completely ordered before \( op_1^j(y, u) \) in \( \tau_i \). If there is more than one ordered pair that share the same second operation (namely \( op_1^j \)), then choose the pair whose first operation is latest.

Since \( \tau_i \) is minimal, \( op_1^j(x, v) \) and \( op_1^j(y, u) \) are switched in order to preserve legality.

Since the operations are by the same process \( p_j \), and since the order of operations by the same process to the same object is preserved (last property in Definition 3.3), it follows that \( x \neq y \). Thus, there exists an operation \( op_1^k(x, w) \) which conflicts with \( op_1^j(x, v) \) and \( op_1^j(x, v) \rightarrow \tau_i \).

Note also that there are no strong operations by \( op_1^j \) and \( op_1^k \).

It is also the case that \( j \neq k \). Suppose otherwise. Either \( op_1^j \) and \( op_1^k \) are switched or else \( op_1^j \) and \( op_1^k \) are switched. In the first case, \( op_1^j \) and \( op_1^k \) would have been the pair chosen and in the second case, \( op_1^j \) and \( op_1^k \) would have been the pair chosen.

Since \( j \neq k \), \( op_1^j \) and \( op_1^k \) conflict, and there is no strong operation by \( p_j \) between them, there is a data race between \( op_1^j \) and \( op_1^k \). Thus the claim is shown.

Let \( I \) be the set of operations in \( \tau_i \) that influence either \( op_1^j \) or \( op_1^k \) (including \( op_1^j \) and \( op_1^k \)). Let \( \tau_i' \) be the shortest prefix of \( \tau_i \) that includes every operation in \( I \). Let \( \tau \) be the subsequence of \( \tau_i' \) which contains exactly the set of all operations in \( I \). The following claim shows that \( \tau \) is consistent with \( \tau_i' \), for all \( l \).

**Claim 5.3** For every pair of operations \( op_1^j \) and \( op_1^k \), if \( op_1^j \rightarrow \tau_i \) then \( op_1^j \rightarrow^{fcs} \rightarrow \tau_i \).

**Proof:** By Claim 2.2 and the constraints on \( \tau_i \), the influence relation is consistent with \( \tau_i \).

Therefore, no operation in \( I \) follows \( op_1^k \) in \( \tau_i \), implying that \( op_1^k \) is the last operation in \( \tau_i' \). By Claim 5.2, \( \tau_i' \) does not include any pair of switched operations, and neither does \( \tau \).

Now, we construct a new partially admissible (with respect to the \( fcs_i \)'s) sequence \( \tau' \) by adding to \( \tau \) every operation \( op_1^l \) not in \( \tau \), such that \( op_1^l \rightarrow^{fcs,j} \rightarrow \tau_i \), and \( op_1^l \) is in \( \tau_i \), for each process \( l \), as follows. For every such operation that was originally in \( \tau_i' \), add it in the same place as in \( \tau_i' \). For all other operations, add them arbitrarily, maintaining consistency with \( \tau_i ' \), for all \( l \). By the definition of hybrid consistency, every added operation \( op_1^l \) that is not in \( \tau_i' \) must be weak. This is because \( op_1^l \) precedes \( op_1^k \) in \( fcs_i \), but follows \( op_1^k \) in \( \tau_i \), a possibility precluded of any strong operation by hybrid consistency.
To prove this result, we consider a legal sequence of memory operations $\tau_i$, as guaranteed for some process $p_i$ in the definition of hybrid consistency, that is minimal with respect to the number of switched operations (operations by the same process $p_i$ whose order in $\tau_i$ is not consistent with $p_i$'s flow control sequence). We show that if $\tau_i$ is not fully admissible (i.e., a sequential execution), then there exists a prefix of a sequential execution of the program that contains a data race. If $\tau_i$ is not fully admissible, it must contain at least one pair of switched operations. We locate the “first” pair of switched operations in $\tau_i$, such that no other pair of switched operations is ordered between them. Because $\tau_i$ is minimal we know this pair was switched to preserve legality. This fact is used to show that there is a data race between some pair of operations that precedes this switched pair. Our main problem is to place these two operations (and the 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 $\tau_i$, and adding any operations necessary to preserve the flow control sequences of all processes.

The key point to prove about the resulting sequence is its legality. In doing so, we either change the value that a read returns (and invoke Lemma 2.3), or, if this does not help, we show that there is a data race earlier in the sequence. Thus we have constructed a prefix of a sequential execution with a data race, which is a contradiction. The details follow.

Fix a data-race-free program $\text{Prog}$, a hybrid consistent run $R$ of $\text{Prog}$, and as guaranteed by the definition of hybrid consistency, a subset $S$ of the memory operations and a flow control sequence $fcs_i$ for each process $p_i$.

Fix a particular process $p_i$. Let $T$ be the set of all operation sequences $\tau_i$ that satisfy Definition 3.3 (hybrid consistency). For $\tau_i$ in $T$, an ordered pair of operations of $p_j$, $\langle \text{op}_j^2, \text{op}_j^1 \rangle$, is switched in $\tau_i$ if $\text{op}_j^2 \xrightarrow{\tau_i} \text{op}_j^1$, but $\text{op}_j^1 \xrightarrow{fcs_i} \text{op}_j^2$.

Let $\tau_i$ be some element of $T$ with a minimal set of switches. That is, there does not exist $\tau'_i \in T$, such that the set of pairs of switched operations of all processes in $\tau'_i$ is strictly contained in the set of pairs of switched operations of all processes in $\tau_i$.

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

**Lemma 5.1** If $\tau_i$ is not fully admissible, then there exists a prefix of a sequential execution of $\text{Prog}$ which contains a data race.

**Proof:** Assume that $\tau_i$ is not fully admissible. We first prove the following claim, which locates the pair of operations that is a candidate for a data race.

**Claim 5.2** There exist two operations $\text{op}_j^2(x, v)$ and $\text{op}_j^1(x, w)$ in $\tau_i$ such that

1. $\text{op}_j^2(x, v) \xrightarrow{\tau_i} \text{op}_j^1(x, w)$,
2. there is a data race between $\text{op}_j^2(x, v)$ and $\text{op}_j^1(x, w)$, and
For which we can take:
\[
\begin{align*}
\tau_1 &= w_1^2(x, 2), w_1^1(x, 1), w_1^2(y, 1), w_1^2(y, 2), sr_1^3(y, 2), sr_2^3(x, 1) \\
\tau_2 &= w_1^2(y, 1), w_2^1(y, 2), w_2^3(x, 2), w_1^1(x, 1), sr_1^3(y, 2), sr_3^3(x, 1)
\end{align*}
\]

\(R\) is a hybrid consistent run in which the values which are written are unique, but not every write is read. However, the reader can verify that \(R\) is not sequentially consistent. Therefore, the assumption that each write is read is necessary for the proof of Theorem 4.4.

5 Running Data-Race-Free Programs

In this section we prove that data-race-free programs behave on hybrid consistent memory implementations as if they were sequentially consistent. Hybrid consistency is a weaker condition than sequential consistency, and can be implemented more efficiently [28, 10, 9]. Clearly, having data-race-free programs behave on hybrid consistent memory implementations as if they were sequentially consistent is a desirable property, since many concurrent programs attempt to be data-race-free.

5.1 Definition of Data-Race-Free Programs

Let \(op_1^i\) and \(op_2^j\) be two operations appearing in some sequence of memory operations \(\alpha\). Then

- \(op_1^i \xrightarrow{p_\alpha} op_2^j\) if \(i = j\) and \(op_1^i \xrightarrow{\alpha} op_2^j\).
- \(op_1^i \xrightarrow{s_\alpha} op_2^j\) if both \(op_1^i\) and \(op_2^j\) are strong operations and \(op_1^i \xrightarrow{\alpha} op_2^j\).

The relation \(\text{happens before}\), denoted by \(\xrightarrow{hb}\), is the transitive closure of the union of \(\xrightarrow{p_\alpha}\) and \(\xrightarrow{s_\alpha}\).

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

Definition 5.1 A program is data-race-free if none of its sequential executions contains a data race.

5.2 Running Data-Race-Free Programs on Hybrid Consistent Memory

We prove that every hybrid consistent run of a data-race-free program is sequentially consistent.
By construction of $\tau$, $w_j(x, w)$ comes before $r_m(x, w)$ in $\tau$. By hybrid consistency, $r_m(x, w)$ comes before $r_i(x, v)$ in $\tau$ since the reads are strong. I.e.,

$$\tau : \ldots w_j(x, w) \ldots r_m(x, w) \ldots r_i(x, v) \ldots$$

By hybrid consistency, $r_m(x, w)$ also comes before $r_i(x, v)$ in $\tau_k$. Since $\tau_k$ is legal and every write is unique (assumption (a)), we also get:

$$\tau_k : \ldots w_j(x, w) \ldots r_m(x, w) \ldots w_k(x, v) \ldots r_i(x, v) \ldots$$

Thus by construction of $\tau$, $w_k(x, v)$ comes between $r_m(x, w)$ and $r_i(x, v)$ in $\tau$. I.e.,

$$\tau : \ldots w_j(x, w) \ldots r_m(x, w) \ldots w_k(x, v) \ldots r_i(x, v) \ldots$$

A contradiction. $\blacksquare$

We now give two counter-examples to show that the assumptions made in order to prove Theorem 4.4 are necessary. The first example demonstrates that the theorem is incorrect without the assumption that the values written to the same location are unique. The second example demonstrates that the theorem is incorrect without the assumption that every write is read.

For the first counter-example, consider the following run $R$, assuming the initial value of every object is 0:

$$R|p_1 : w_1^1(y, 1), w_1^2(x, 2)$$
$$R|p_2 : w_2^1(y, 1), w_2^2(x, 2)$$
$$R|p_3 : sr_3^1(x, 2), sr_3^2(y, 0), sr_3^2(y, 1), sr_3^4(y, 1), sr_3^5(x, 2)$$

For which we can take:

$$\tau_1 = w_2^2(x, 2), sr_3^1(x, 2), sr_3^2(y, 0), w_1^1(y, 1), sr_3^3(y, 1), w_1^4(y, 1), sr_3^4(y, 1), w_1^5(y, 1), sr_3^5(x, 2)$$

$$\tau_2 = w_1^3(x, 2), sr_3^1(x, 2), sr_3^3(y, 0), w_1^1(y, 1), sr_3^3(y, 1), w_1^4(y, 1), sr_3^4(y, 1), w_1^5(y, 1), sr_3^5(x, 2)$$

$$\tau_3 = w_1^3(x, 2), sr_3^1(x, 2), sr_3^3(y, 0), w_1^1(y, 1), sr_3^3(y, 1), w_1^4(y, 1), sr_3^4(y, 1), w_1^5(y, 1), sr_3^5(x, 2).$$

$R$ is a hybrid consistent run in which every write is read, but the values which are written are not unique. However, $R$ is not a sequentially consistent run since $sr_3^1(x, 2)$ means that either $p_1$ or $p_2$ has already written to both $y$ and $x$, and it’s not possible that $r_3^1(y, 0)$ returns the initial value of $y$. Therefore, the assumption that the values which are being written are unique is necessary for the proof of Theorem 4.4.

For the second counter-example, consider the following run $R$, assuming the initial value of every object is 0:

$$R|p_1 : w_1^1(x, 1), w_1^2(y, 1), sr_3^3(y, 2)$$
$$R|p_2 : w_2^1(y, 2), w_2^2(x, 2), sr_3^3(x, 1)$$

19
However, in $\tau_0$, $p_0$ continuously wants to enter the critical section while in $\tau_1$, $p_1$ never enters the critical section and in $\tau_2$, $p_2$ never enters the critical section, indicating a deadlock.

### 4.3 Strong Reads

We now prove that labeling all reads as strong suffices for providing sequential consistency. The proof of this theorem relies on the following assumptions about the run: (a) every value written to the same object is unique, and (b) every value written is returned by some read. We show by specific counter-examples that both these assumptions are necessary to prove the above result.

The unique writes assumption (Assumption (a)) is often made for proving properties about various consistency conditions [22, 30]; it is sometimes regarded as being merely technical, made in favor of simplicity and having no effect of the correctness of the claims. Our counter-example indicates that this assumption is not merely technical, and that special care should be taken whenever it is made.

**Theorem 4.4** Every hybrid consistent run of a program in which all reads are strong and all writes are weak is sequentially consistent.

**Proof:** The proof is similar to that of the theorem for strong writes.

Let $R$ be such a run. Let $S$ be a subset of memory operations in $R$, $fcs_i$ for each $p_i$ be a flow control sequence, $\rho$ be a permutation of the strong operations (namely, the reads) in $S$, and $\tau_i$ for each $p_i$ be a legal permutation of $S$ as guaranteed by the definition of hybrid consistent. We will construct $\tau$, a legal permutation of $S$ that is fully admissible with respect to the $fcs_i$'s.

The method for constructing $\tau$ is to place the reads as in $\rho$, and to place each write by $p_i$ in the interval (between reads) where it appears in $\tau_i$. All the writes that end up in the same interval can be ordered in any way, as long as the $fcs_i$ orders are preserved.

The fact that $\tau$ respects the $fcs_i$'s follows from the construction of $\tau$ and the fact that $\tau_i$ respects $p_i$'s ordering.

We now show that $\tau$ is legal. Assume, by way of contradiction, that $\tau$ is not legal. So there is some read of $x$ (by $p_i$) that returns $v$, where the most recent write to $x$ (by $p_j$) was for $w \neq v$.

$$\tau = \ldots w_j(x, w) \ldots (\text{no write to } x) \ldots r_i(x, v) \ldots$$

By construction of $\tau$, $w_j(x, w)$ comes before $r_i(x, v)$ in $\tau_i$. Since $\tau_i$ is legal, some $w_k(x, v)$ comes between $w_j(x, w)$ and $r_i(x, v)$ in $\tau_i$. Since every value written is read (assumption (b)) and $\tau_j$ is legal, some $r_m(x, w)$ comes between $w_j(x, w)$ and $w_k(x, v)$ in $\tau_j$, i.e.,

$$\tau_j : \ldots w_j(x, w) \ldots r_m(x, w) \ldots w_k(x, v) \ldots r_i(x, v) \ldots$$


Entry: wait until read($turn[i]$);

(critical section)

Exit: write($turn[i]$,false);
    write($turn[i+1]$,true);

Remainder:
    every so often do
        if read($turn[i]$) then
            write($turn[i-1]$,false);
            write($turn[i]$,false);
            write($turn[i+1]$,true);
        endif
    enddo;

Figure 2: A cooperative solution for the mutual exclusion problem – code for $p_i$

These operations are from $p_1$’s remainder section; after once transferring the token to $p_2$, it continuously reads false from its turn variable (indicated by the ellipses). In this run, $p_2$’s operations, in the flow control sequence, are

$$r_1^1(turn[2],true) \ w_3^2(turn[1],false) \ w_3^3(turn[2],false) \ w_3^4(turn[0],true) \ r_1^1(turn[2],false)\ldots$$

These operations are from $p_2$’s remainder section; after once transferring the token to $p_3$, it continuously reads false from its turn variable (indicated by the ellipses).

Recall that all operations under consideration are weak. The following $\tau_i$'s demonstrate that this run is hybrid consistent (the ellipses indicate the continuous reads of false from the $turn$ variables):

$$\tau_0 : r_3^1(turn[0],true) \ w_3^2(turn[2],false) \ w_3^3(turn[0],false) \ w_3^4(turn[1],true) \ r_3^1(turn[1],true) \ w_3^4(turn[2],true) \ r_3^1(turn[2],true) \ w_3^4(turn[0],true) \ w_3^4(turn[0],false) \ w_3^2(turn[1],false) \ w_3^2(turn[2],false)\ldots$$

$$\tau_1 : r_3^1(turn[0],true) \ w_3^2(turn[1],true) \ r_3^1(turn[1],true) \ w_3^4(turn[0],false) \ w_3^4(turn[1],false) \ w_3^4(turn[2],true) \ r_3^1(turn[2],true) \ w_3^4(turn[0],true) \ w_3^4(turn[2],false) \ w_3^4(turn[0],false) \ w_3^2(turn[1],false) \ w_3^2(turn[2],false)\ldots$$

$$\tau_2 : r_3^1(turn[0],true) \ w_3^2(turn[1],true) \ r_3^1(turn[1],true) \ w_3^4(turn[2],true) \ r_3^1(turn[2],true) \ w_3^4(turn[0],true) \ w_3^4(turn[2],false) \ w_3^4(turn[1],false) \ w_3^4(turn[2],false) \ w_3^4(turn[0],true)$$
Since the exit points are strong writes, all the processes agree (in the \( \tau_i \)'s) on the same non-overlapping order for the critical pairs.

Let \( \tau \), \( \tau' \) and \( \tau'' \) be three critical pairs such that \( \tau \) is ordered before \( \tau' \) and after \( \tau'' \) in any \( \tau_i \). Since the exit point of \( \tau \) is a strong operation, and since it is ordered before the entry point of \( \tau' \), all the operations of \( \tau \) are ordered before the entry point of \( \tau' \). Since the entry point of \( \tau \) is a strong operation, and since it is ordered after the exit point of \( \tau'' \), all the operations of \( \tau \) are ordered after \( \tau'' \). Thus, (logical) mutual exclusion is also guaranteed by the algorithm.

Several papers on relaxed consistency conditions refer to the above method of programming with critical sections as a justification for the separation of strong and weak operations. For example, in [19], it is argued (page 19):

“For example, a large class of programs are written such that accesses to shared data are protected within critical sections. Such programs are called synchronized programs, whereby writes to shared locations are done in a mutually exclusive manner (no other reads or writes can occur simultaneously). In a synchronized program, all accesses (except accesses that are part of the synchronization constructs) can be labelled as ordinary.”

While Theorem 4.3 supports this claim, one should note the extra care required in formally proving it. In particular, we had to add an extra write in the entry section. Moreover, the theorem is not valid if the mutual exclusion algorithm is cooperative, as we show now.

Note that, in cooperative solutions, a process that participates in the mutual exclusion protocol, but does not wish to enter the critical section, can access variables in the exclusion set while executing the remainder section. If these accesses are labeled as weak, the solution might not be correct anymore.

For example, consider the cooperative solution in Figure 2. It can be seen that the algorithm solves mutual exclusion in any sequentially consistent run. Essentially, the variable \( \text{turn} \) represents a token that circulates around (it is always located at the process \( p_i \) with \( \text{turn}[i] = \text{true} \)).

We now describe a hybrid consistent run for three processes, \( p_0 \), \( p_1 \) and \( p_2 \), in which the token is “lost” and a deadlock is created. In this run, \( \text{turn}[0] \) is initially true, \( \text{turn}[1] \) and \( \text{turn}[2] \) is are initially false and \( p_0 \)'s operations, in the flow control sequence, are:

\[
\begin{align*}
& r_0^4(\text{turn}[0],\text{true}) \quad w_0^3(\text{turn}[2],\text{false}) \quad w_0^3(\text{turn}[0],\text{false}) \quad w_0^4(\text{turn}[1],\text{true}) \quad r_0^5(\text{turn}[0],\text{false})
\end{align*}
\]

The first four operations are from \( p_0 \)'s remainder section; then \( p_0 \) reads false from \( \text{turn}[0] \) in its entry section forever (indicated by the ellipses). In this run, \( p_1 \)'s operations, in the flow control sequence, are

\[
\begin{align*}
& r_1^4(\text{turn}[1],\text{true}) \quad w_1^3(\text{turn}[0],\text{false}) \quad w_1^3(\text{turn}[1],\text{false}) \quad w_1^4(\text{turn}[2],\text{true}) \quad r_1^5(\text{turn}[1],\text{false})
\end{align*}
\]
Thus, the set of sequences \( \{\tau_j\}_{j=1}^n \) that results from eliminating all the operations that do not appear in \( R' \) from \( \{\tau_j\}_{j=1}^n \) obeys all the requirements in the definition of hybrid consistency (w.r.t. the \( \{\text{exc}'\}_{j=1}^n \)). Note that all the writes in \( R' \) are strong, and thus, by Theorem 4.1, \( R' \) is sequentially consistent. Furthermore, in \( R' \) there is a deadlock. Since there are no reads from \( \text{nac}(A) \), the run \( R'' \) that results from eliminating all the writes to \( \text{nac}(A) \) is also sequentially consistent and there is a deadlock in \( R'' \). Note that \( R'' \) is a run of \( A \) in the case of an empty critical section. Thus, there is a sequentially consistent run of \( A \) that has a deadlock. A contradiction.

We now show that the algorithm provides (logical) mutual exclusion. Note that, since \( A \) guarantees mutual exclusion based on sequential consistency, every sequential execution of \( A \) must guarantee mutual exclusion. We will show that if \( A' \) does not guarantee mutual exclusion based on hybrid consistency, we may build a sequential execution of \( A \) that does not guarantee mutual exclusion.

Let \( R \) be a hybrid consistent run of \( A' \), and let \( \{\tau_j\}_{j=1}^n \) be the set of sequences of operations as guaranteed in the definition of hybrid consistency. Call the last operation executed by a process before entering the critical section the *entry point*. Note that the entry point is a write operation. By Claim 4.2, there is at least one write in the exit section. Call the first write in the exit section the *exit point*. Each pair of matching entry and exit points is called a *critical pair*. We claim, that in each \( \tau_j \), all the critical pairs are ordered in a non-overlapping way.

Assume, by way of contradiction, that there exists a sequence \( \tau_j \) in which two critical pairs do overlap. Denote two of these pairs by \([op_k^1, op_k^2] \) and \([op_l^1, op_l^2] \) and assume, without loss of generality, that \( op_k^1 \tau op_k^2 \). Let \( R' \) be the run that results by eliminating from \( R \) all operations that are not invoked inside entry and exit sections of \( A' \). Since the algorithm is non-cooperative, \( R' \) includes all the writes (in \( R \)) to variables in \( \text{exc}(A') \), and thus, \( R' \) is hybrid consistent. By Theorem 4.1, \( R' \) is sequentially consistent. Thus, we may build a legal sequence of operations \( \tau \) in the same way as in the proof of Theorem 4.1. That is, the strong writes are ordered in \( \tau \) in the order they appear in any sequence \( \tau_j \). A read by process \( p_i \) is inserted after any write that precedes it in \( \tau_j \) and before any write that follows it in \( \tau_j \). Furthermore, it will be inserted after any read by \( p_i \) that precedes it in \( \tau_j \) and before any read by \( p_i \) that follows it in \( \tau_j \). The ordering of read operations by different processes in unimportant. Note that \( \tau \) is a sequential execution of \( A' \) in the case of an empty critical section.

By the construction of \( \tau \), and since we assumed that \([op_k^1, op_k^2] \) and \([op_l^1, op_l^2] \) are overlapping in \( \tau_j \), the latest of \( op_k^1 \) and \( op_l^1 \) precedes both \( op_k^2 \) and \( op_l^2 \) in \( \tau \). Consider the prefix \( \tau' \) of \( \tau \) that ends with the latest of \( op_k^1 \) and \( op_l^1 \). Note that any operation in \( \tau \) that is invoked during the exit sections that corresponds to \([op_k^1, op_k^2] \) and \([op_l^1, op_l^2] \) is a read. Remember, that there are no reads from \( \text{nac}(A) \). Thus, \( \tau'' \), the result of eliminating from \( \tau' \) all reads that are invoked during the exit sections that corresponds to \([op_k^1, op_k^2] \) and \([op_l^1, op_l^2] \) and all writes to \( \text{nac}(A) \), is a prefix of a sequential execution of \( A \). Moreover, at the end of \( \tau'' \), both \( p_k \) and \( p_l \) are inside the critical section. Thus, \( \tau'' \) can be extended to a sequential execution of \( A \) that does not guarantee mutual exclusion. A contradiction to the assumption that every sequential execution of \( A \) guarantees mutual exclusion.
Recall that $tmp$ and $x$ are not accessed in the entry and exit sections of $A$. Consider now a sequentially consistent run $R$ of $A$ with this critical section in which $p_0$ starts at time 0 and runs until it enters the critical section, completes it, and exits; then $p_1$ starts and runs until it enters the critical section, completes it, and exits. Denote by $t$ the time at which $p_0$ completes the read operation from $x$.

We claim that there is a sequentially consistent run $R'$ of $A$ with this critical section in which $p_0$ starts at time 0 and runs until time $t$, when it completes the read operation; then $p_1$ starts and runs until it enters the critical section, completes it, and exits; then $p_0$ completes its critical section and exits. Note that there is no write in the exit section of $A$ and there is no write to variables in $exc(A)$ in the critical section of $p_0$. Thus, all the values read by $p_1$ in its entry section in $R'$ are equal to the values it reads in its entry section in $R$; since in $R$, $p_1$ must eventually enter the critical section (to avoid deadlock) it must eventually enter the critical section in $R'$.

Clearly, $R'$ violates the standard definition of mutual exclusion, since $p_0$ and $p_1$ are inside the critical section at the same time. Furthermore, note that in $R'$ both $p_0$ and $p_1$ read the value 0 from $x$ and thus the final value of $x$ at the end of $R'$ is 1. This value could not happen in any run that preserves logical mutual exclusion.

Given a non-cooperative algorithm $A$ for the mutual exclusion problem based on sequential consistency, label every write in the entry and exit sections as strong and every read in the entry and exit sections as weak; operations inside the critical section are labeled as weak. Next, add a strong write to some object $nac(A)$ that is not accessed elsewhere in the program such that this write will be the last instruction executed (in a now control sequence) before each critical section. Call this modified algorithm $A'$. We have:

**Theorem 4.3** $A'$ guarantees mutual exclusion based on hybrid consistency.

**Proof:** In order to prove that $A'$ guarantees mutual exclusion based on hybrid consistency, we have to show that it guarantees mutual exclusion and that it is free of deadlocks.

Assume, by way of contradiction, that there exists a hybrid consistent run $R$ of $A'$ that has a deadlock. Denote by $\{ f_{i,j} \}_{i,j=1}^n$ the set of sequences of operations as guaranteed by the definition of hybrid consistency. Let $R'$ be the run that results by eliminating from $R$ all operations that are not invoked inside (w.r.t. the flow control sequences) the entry or exit section of $A'$. Since the algorithm is non-cooperative, there are no writes to variables in $exc(A')$ outside (w.r.t. the flow control sequences) the entry and exit sections of $A'$. Thus, $R'$ includes all the writes (in $R$) to variables in $exc(A')$, and thus, $R'$ is hybrid consistent. Specifically, define for each $j$ a flow control sequence $fcs_j$ by eliminating from $fcs_j$ all the operations of $p_j$ that do not appear in
Let $R$ be a hybrid consistent run of $\mathcal{A}$ and let $\{\tau_i\}_{i=1}^n$ be a set of sequences as required in the definition of hybrid consistency. Consider the $cs_k^i$'s induced by the flow control sequences from the definition of hybrid consistency. $\mathcal{A}$ guarantees logical mutual exclusion based on hybrid consistency provided that:

**Mutual exclusion:** For any four operations $op_1^i, op_2^i \in cs_k^i$ and $op_1^j, op_2^j \in cs_l^j$, $op_1^i \xrightarrow{\tau^*} op_2^i$ if and only if $op_1^j \xrightarrow{\tau^*} op_2^j$ for every $p$ and $q$. As before, this implies that there is a total order on all critical section executions; furthermore, this ordering is the same in all $\tau_i$'s.

**Deadlock freedom:** If $\tau_i$ is infinite and there exists a process $p_i$ that is in its entry section from some point on in $\tau_i$, then there is another process $p_j$ that enters (and leaves) its critical section infinitely often in $\tau_j$.

In the rest of this section we mean logical mutual exclusion whenever we refer to mutual exclusion.

Our result assumes that the mutual exclusion algorithm does not communicate with the algorithm that is executed inside the critical section and in the remainder section. Specifically, we assume that variables that are accessed in the entry or exit section are not accessed in the critical or remainder sections. To capture this property, let the exclusion set of a mutual exclusion algorithm $\mathcal{A}$ be the set of shared variables read in the entry or exit sections of $\mathcal{A}$; this set is denoted $exc(\mathcal{A})$.

**Definition 4.1** A mutual exclusion algorithm $\mathcal{A}$ is non-cooperative if every process which executes the critical section or the remainder section of $\mathcal{A}$ does not write any variable in $exc(\mathcal{A})$; otherwise, the algorithm is cooperative.\(^5\)

We assume that the mutual exclusion algorithm is designed to run with any code in the critical section (except the above restriction), and that it can be run on asynchronous systems.

**Claim 4.2** There is at least one write to a variable in $exc(\mathcal{A})$ in the exit section of every non-cooperative mutual exclusion algorithm based on sequential consistency.

**Proof:** Assume, by way of contradiction, that there exists a non-cooperative mutual exclusion algorithm $\mathcal{A}$ such that there is no write to any of the variables of $exc(\mathcal{A})$ in the exit section of $\mathcal{A}$. Assume that we run the algorithm with the following critical section (for every process $p_i$), under the assumption that $x$ is initially 0:

\[ \begin{array}{c}
\text{Claim 4.2} \quad \text{There is at least one write to a variable in } exc(\mathcal{A}) \text{ in the exit section of every non-cooperative mutual exclusion algorithm based on sequential consistency.}
\end{array} \]

\[ \begin{array}{c}
\text{Proof:} \quad \text{Assume, by way of contradiction, that there exists a non-cooperative mutual exclusion algorithm } \mathcal{A} \text{ such that there is no write to any of the variables of } exc(\mathcal{A}) \text{ in the exit section of } \mathcal{A}. \text{ Assume that we run the algorithm with the following critical section (for every process } p_i), \text{ under the assumption that } x \text{ is initially 0:}
\end{array} \]

\[ \begin{array}{c}
\text{5 The definition of non-cooperative algorithms given in [9] allows processes that are executing the critical section to access variables in } exc(\mathcal{A}). \text{ The additional requirement made here seems quite reasonable, since we usually want to treat the mutual exclusion algorithm as a general solution, independent of the rest of the code, and to use it as a subroutine. This is also the approach taken in many of the known solutions to the mutual exclusion problem [36]. A more detailed discussion and motivation for non-cooperative mutual exclusion algorithms appears in [9].}
\end{array} \]

13
weak. We demonstrate this method for mutual exclusion. Given a mutual exclusion algorithm designed for sequentially consistent memories, we produce a modified algorithm by adding one strong write (to a location which is never read from) to the entry section and by labeling all the writes in the synchronization part of the code as strong, while all other operations are labeled as weak. We prove that the modified algorithm guarantees mutual exclusion (in a strong sense) on hybrid consistent memories.

We remind the reader that an algorithm for mutual exclusion consists of four disjoint sections for each process—entry, critical, exit and remainder (cf. [36]). In the entry section, a process tries to gain access to the critical section; the exit section is executed by each process upon leaving the critical section; the remainder section is the rest of the code. Processes cycle through the four sections of their code. An algorithm guarantees mutual exclusion provided that:

**Mutual exclusion:** no two processes are inside the critical section at the same time, and

**Deadlock freedom:** in every infinite run, if there exists a process that executes the entry section from some point on in this run, then there exists another process that enters (and leaves) the critical section infinitely often during this run.

A mutual exclusion algorithm provides code for the entry and exit sections. It should treat the code in the critical and remainder sections as black boxes (with some restrictions, as discussed below).

The above definition of mutual exclusion assumes that the order operations are executed reflects the order they are viewed by the processes. However, our formalism for defining consistency conditions puts restrictions only on the order operations are viewed, which may not correspond directly to the order they are executed. To be able to cope with these conditions, we define **logical mutual exclusion** as follows. Given a mutual exclusion algorithm (program), consider a flow control sequence for process $p_i$ and let $cs^k_i$ be the set of operations invoked by process $p_i$ during the $k$th time $p_i$ executes the critical section in this sequence.

Let $R$ be a sequentially consistent run of $A$ and let $\tau$ be a sequence of memory operations as required in the definition of sequential consistency. Consider the $cs^k_i$'s induced by the flow control sequences from the definition of sequential consistency. $A$ guarantees **logical mutual exclusion based on sequential consistency** provided that:

**Mutual exclusion:** For every four operations $op^1_i, op^2_i \in cs^k_i$ and $op^1_j, op^2_j \in cs^l_j$, $op^1_i \rightarrow op^1_j$ if and only if $op^2_i \rightarrow op^2_j$. That is, all operations executed inside one critical section are ordered before all the operations in the other critical section. Note that this implies that there is a total order on all critical section executions.

**Deadlock freedom:** If $\tau$ is infinite and there exists a process that is in its entry section from some point on in $\tau$, then there exists another process that enters (and leaves) its critical section infinitely often in $\tau$. 
4 Static Approach

In this section we discuss techniques for writing programs for hybrid consistent shared memories that are based on statically classifying accesses depending on whether they are reads or writes. In Section 4.1, we show that every hybrid consistent run of a program in which all writes are strong is sequentially consistent. This result is used to develop efficient synchronization code, in Section 4.2. We then show symmetrically, in Section 4.3, that every hybrid consistent run of a program in which all reads are strong is sequentially consistent. In order to prove this theorem, we make further assumptions on the run and demonstrate their importance.

4.1 Strong Writes

We prove the following theorem:

**Theorem 4.1** Every hybrid consistent run of a program in which all writes are strong and all reads are weak is sequentially consistent.

**Proof:** Let $R$ be such a run. Let $S$ be a subset of memory operations in $R$, $fcs_i$ for each $p_i$ be a flow control sequence, $\rho$ be a permutation of the strong operations (namely, the writes) in $S$, and $\tau_i$ for each $p_i$ be a legal permutation of $S$ as guaranteed by the definition of hybrid consistency. We will insert the (weak) read operations into $\rho$ to construct $\tau$, a legal permutation of $S$ that is fully admissible with respect to the $fcs_i$'s.

A read by process $p_i$ is inserted after any write that precedes it in $\tau_i$ and before any write that follows it in $\tau_i$. This can be done since $\tau_i$ agrees with $\rho$ on the order of strong operations. Furthermore, it will be inserted after any read by $p_i$ that precedes it in $\tau_i$ and before any read by $p_i$ that follows it in $\tau_i$. The ordering of read operations by different processes is unimportant.

Clearly, $\tau$ is a permutation of $S$. Since $\tau_i$ preserves the ordering of operations by $p_i$, it follows that $\tau$ is $fcs_i$-admissible for all $i$.

The fact that $\tau$ is legal follows from the fact that $\tau_i$ is legal, that $\tau_i$ agrees with $\rho$ on the order of strong operations and from the construction of $\tau$.

4.2 Using Strong Writes to Program with Critical Sections

Theorem 4.1 is very useful for designing and proving correctness of programs which rely on hybrid consistency. A simple way to use it is to take a program which is designed for sequential consistency, label each write as a strong write and each read as a weak read, and run it on a hybrid consistent memory. However, there is even a more efficient way to employ the above theorem. If the program has explicit synchronization code dedicated to coordinating memory access operations, while the rest of the code ignores synchronization issues, then it is possible to apply Theorem 4.1 only to the synchronization code, and label all other memory accesses as
We now discuss the last property in more detail. It states that all operations on the same object by the same process $p_j$ are viewed by every other process in the same order as they are viewed by $p_j$. This property does not appear in the original definition of hybrid consistency [9]. However, it is necessary in order for some of our results to hold, as is shown in Section 5.3. Evidence suggests it is a reasonable assumption, since some previous authors make the even stronger assumption that all processes view all operations on the same object, no matter which process invoked them, in the same order.\footnote{In [2, 20, 21], a total order on all the writes to the same location is assumed. In addition, it is assumed that a value read from a specific location can be uniquely identified with a write operation. Thus, all the processes view all the writes to the same location in the same order. Since a value read from a specific location can be uniquely identified with a write operation, each read is viewed by all the processes to be between the same writes to that location. These two assumptions imply that all the processes view all the operations on the same object in the same order.}

### 3.1 Ignoring Control Operations

To illustrate the problem that occurs if hybrid consistency were defined without considering control operations, consider the following example, assuming $x$ and $y$ are initially 0 and all instructions are weak:

<table>
<thead>
<tr>
<th>$p_1$'s program</th>
<th>$p_2$'s program</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{tmp}_1 := \text{read}(x)$;</td>
<td>$\text{tmp}_2 := \text{read}(y)$;</td>
</tr>
<tr>
<td>if $\text{tmp}_1 = 5$ then</td>
<td>if $\text{tmp}_2 = 5$ then</td>
</tr>
<tr>
<td>$\text{write}(y, 5)$;</td>
<td>$\text{write}(x, 5)$;</td>
</tr>
</tbody>
</table>

In this program, 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. For example, assume this program is being executed on a hybrid consistent memory, and all operations are weak. If we ignore the existence of control instructions, the definition would allow the following run $R$:

| $R|p_1$ | $r_1^1(x, 5), w_1^2(y, 5)$ |
|--------|--------------------------|
| $R|p_2$ | $r_2^1(y, 5), w_2^2(x, 5)$ |

for which we can take:

$$
\tau_1 = w_2^2(x, 5), r_1^1(x, 5), w_1^2(y, 5), r_2^1(y, 5)
$$

$$
\tau_2 = w_1^2(y, 5), r_2^1(y, 5), w_2^2(x, 5), r_1^1(x, 5)
$$

The program for $p_1$ allows the $\text{read}(x)$ to return 5, whereas $x$ could have taken the value 5 only if $p_1$ writes 5 to $y$, which will happen only if $p_1$ reads 5 in $x$. 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. If such behavior is allowed, then writing programs and arguing about them becomes almost impossible.
It requires that there exist a subset of the memory operations in the run and a set of flow control sequences such that for each process, there is a legal permutation of those operations that is consistent with both the process' flow control sequence and every other process' control order. (Weak consistency may appear at first glance to be incomparable with sequential consistency, but it is strictly weaker, since the second property is implied by the sequential consistency condition.)

**Definition 3.2 (Weak consistency)** A run \( R \) is weakly consistent if there exists a subset \( S \) of the memory operations in \( R \), a set of flow control sequences \( \{ fcs_i \}_{i=1}^n \), one for each \( p_i \), such that for each \( p_i \), there exists a legal permutation \( \tau_i \) of \( S \) with the following properties.

1. \( \tau_i \) is fully \( fcs_i \)-admissible.
2. If \( op_1^j \xRightarrow{fcs_i} op_2^j \), then \( op_1^j \xRightarrow{\tau_i} op_2^j \), for any \( j \).

Hybrid consistency [9] is intermediate between sequential and weak consistency: it combines the expressiveness of the former and the efficiency of the latter. Hybrid consistency distinguishes between two types of operations—strong and weak. It states that there must be a subset of the memory operations in the run, a total order on the strong operations among them, and a set of flow control sequences satisfying the following: For each process, there is a legal permutation of the operations in the subset that is consistent with four orders: the process' flow control sequence, every process' control order, the total order on the strong operations, and the relative order of every pair of strong and weak operations by another process in that process' flow control sequence. Furthermore, all accesses of the same process to the same location will be viewed by all the processes in the same order. It is possible to implement hybrid consistency in such a way that weak operations are extremely fast [9].

**Definition 3.3 (Hybrid consistency)** A run \( R \) is hybrid consistent if there exists a subset \( S \) of the memory operations in \( R \), a set of flow control sequences \( \{ fcs_i \}_{i=1}^n \), and a permutation \( \rho \) of the strong operations in \( S \) such that for each process \( p_i \), there exists a legal permutation \( \tau_i \) of \( S \) with the following properties:

1. \( \tau_i \) is fully \( fcs_i \)-admissible.
2. If \( op_1^j \xRightarrow{fcs_i} op_2^j \), then \( op_1^j \xRightarrow{\tau_i} op_2^j \), for any \( j \).
3. If \( op_1^j \xRightarrow{fcs_i} op_2^j \) and at least one is strong, then \( op_1^j \xRightarrow{\tau_i} op_2^j \), for any \( j \).
4. If \( op_1^j \xRightarrow{\rho} op_k^j \) (implying both are strong), then \( op_1^j \xRightarrow{\tau_i} op_k^j \), for any \( j \) and \( k \).
5. If \( op_1^j \xRightarrow{fcs_i} op_2^j \) and \( op_1^j \) and \( op_2^j \) access the same location, then \( op_1^j \xRightarrow{\tau_i} op_2^j \), for any \( j \).
Note that an operation directly influences another operation only if it is ordered before it in \( \tau \). Since the influence relation is the transitive closure of direct influence, we have:

**Claim 2.2** If \( \text{op}_j^1 \) influences \( \text{op}_k^2 \) in \( \tau \), then \( \text{op}_j^1 \xrightarrow{\tau} \text{op}_k^2 \).

The following lemma captures the intuition that if a read operation \( \text{op}_j^1 = r_j(x,v) \) does not influence operation \( \text{op}_k^2 \), then \( \text{op}_k^2 \) would have been generated even if \( \text{op}_j^1 \) had read a different value than \( v \).

**Lemma 2.3** Let \( \tau \) be a sequence of memory operations that is partially admissible with respect to a set of flow control sequences \( \{fcs_i\}_{i=1}^n \). Let operation \( \text{op}_j^1 \) in \( \tau \) be a read \( r_j(x,v) \) that does not influence any operation in \( \tau \). Let \( \tau' \) be the result of taking \( \tau \) and changing \( \text{op}_j^1 \) to be \( r_j(x,w) \) for some \( w \neq v \). Then \( \tau' \) is partially admissible for some set of flow control sequences \( \{fcs_i\}_{i=1}^n \).

### 3 Defining Non-Sequential Consistency Conditions

In this section, we define three consistency conditions that generalize previously known ones for the non-sequential case; a fourth condition, asymmetric hybrid consistency, is presented in Section 6.1. The reason they are generalizations is that in non-optimized systems, where operations at each process are invoked in program order and only one operation may be pending at a time, \( fcs_i \) is simply the sequence of operations in the order they were invoked. We then discuss (by an example) the importance of considering control operations (e.g., if-statements) when specifying consistency conditions. All our definitions focus on the memory operations that appear in the flow control sequences, and ignore the other operations.

Sequential consistency [27] is a strong consistency condition stating that there exists a sequential execution that is consistent with the way the actual run appears to every process. Sequential consistency allows one to reason about a concurrent system using familiar uniprocessor techniques. Since uniprocessor systems are easier to reason about than concurrent systems, this is helpful. Thus many systems (try to) provide sequential consistency. Later in this paper, we show several programming techniques for achieving sequential consistency when the system only provides a weaker condition. Providing sequential consistency in message-based systems requires the response time of some operations to depend on the end-to-end message delay [28, 10].

**Definition 3.1 (Sequential consistency)** A run \( R \) is sequentially consistent if there exists a subset \( S \) of the memory operations in \( R \), a set \( \{fcs_i\}_{i=1}^n \) of flow control sequences, and a legal permutation \( \tau \) of \( S \) such that \( \tau \) is fully admissible with respect to \( \{fcs_i\}_{i=1}^n \).

Weak consistency [28, 9] is a very weak condition; it does not impose any global ordering on any kind of operations. Its importance lies in the fact that it may be implemented very efficiently, and despite its weakness, there is a large class of programs for which it is sufficiently expressive.
2.3 Weak and Strong Operations

It is possible to mark some instructions in a program as strong; all other instructions are weak. An instance of a strong instruction is a strong operation and an instance of a weak instruction is a weak operation. In the case of read/write objects this means that it is possible to use strong reads, strong writes, weak reads, and weak writes. In the rest of the paper, we use \( op_i \) to denote an operation invoked by process \( p_i \) (weak or strong), and by \( sop_i \) we denote a strong operation invoked by process \( p_i \). We use superscripts, e.g., \( op_i^1, op_i^2, \ldots \), to distinguish between operations invoked by the same process (note that the superscript does not imply any ordering of the operations). We sometimes use a shorthand notation for read and write operations and denote by \( r_i(x, v) \) a read operation (weak or strong) invoked by process \( p_i \) returning \( v \) from \( x \); we denote by \( w_i(x, v) \) a write operation (weak or strong) invoked by process \( p_i \) writing \( v \) to \( x \). Similarly, \( sr_i(x, v) \) is a strong read operation invoked by process \( p_i \) returning \( v \) from \( x \); \( sw_i(x, v) \) is a strong write operation invoked by process \( p_i \) writing \( v \) to \( x \).

2.4 Control Operations and the Influence Relation

When defining and analyzing consistency conditions, it is important to take into account the effects of control operations. A specific example of the pitfalls associated with failing to do so appears in Section 3, when we present our new consistency conditions. To capture the effect of control operations, we define a partial order on operations in a flow control sequence; this partial order is featured in the consistency conditions presented below. Based on the relation \( fcs \) we define a partial order \( \Rightarrow \) called the control order. Formally, for any two memory operations \( op_i^1 \) and \( op_i^2 \), \( op_i^1 \Rightarrow op_i^2 \) if there exists a control operation \( op_i^3 \) such that \( op_i^1 \xrightarrow{fcs} op_i^3 \xrightarrow{fcs} op_i^2 \).

We now formalize the notion of one operation influencing another, which relies on the control order. Let \( \tau \) be a sequence of memory operations and for each \( p_i \), let \( co_i \) be a partial order on the operations that is consistent with \( \tau \). An operation \( op_j^1 \) directly influences an operation \( op_k^1 \) in \( \tau \) (with respect to the \( co_i \)'s), if one of the following holds:

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

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

The influence relation is the transitive closure of direct influence. Thus the influence relation is also defined with respect to a set of partial orders. Although we will not usually explicitly mention these partial orders, the influence relation will be used with admissible operation sequences and the relevant partial orders will be the control orders for the corresponding flow control sequences.
2.2 Sequential Executions

Although a run is a sequence of events, it can also be viewed as containing operations. The memory operations in a run are obtained by matching up corresponding call and response events; each control operation is itself an event. We are interested in picking out a subset of the operations in a run, finding an ordering of those operations, and seeing what properties that ordering satisfies. To that end, we next define several properties on sequences of operations. The two main properties are satisfying the serial specifications of the objects (called “legal”) and being consistent with a sequential execution of the program (called “admissible”).

First, a piece of notation: if \( \tau \) is a sequence of operations, and \( op_i \) precedes \( op_j \) in \( \tau \), we write \( op_i \rightarrow op_j \).

A sequence \( \tau \) of operations is *legal* if for each object \( x \), \( \tau|x \), the subsequence of \( \tau \) consisting exactly of the operations involving \( x \), is in the serial specification of \( x \).

The notion of a sequential execution of the program is formalized with the notion of a *flow control sequence* for a process \( p_i \). We build up inductively an execution of \( p_i \)'s program in which every instruction finishes executing before the next one begins. Given process \( p_i \)'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 \( op \), is an instance of instruction \( I \) in the program. If \( op \) 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 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 \( fcs_i \).

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).

**Claim 2.1** Any legal partially admissible sequence of memory operations is a prefix of a sequential execution and vice versa.
methods for programming with hybrid consistency are discussed in Section 4. Data-race-free programs are discussed in Section 5. The results for asymmetric hybrid consistency appear in Section 6. Section 7 compares our results with related work. We conclude with a discussion of the results in Section 8.

2 Framework

In this section, we present our formal definitions.

2.1 System Components

Each 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 process' 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 process at each node as well as possibly other hardware. Every object is assumed to have a serial specification (cf. [25]) 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. For example, in the case of a read/write object, the ordered pair [Read$_i$(x), Return$_i$(x, v)] forms an operation ($p_i$ reads v from x) for any $p_i$, x, and v, as does [Write$_i$(x, v), Ack$_i$(x)] ($p_i$ writes v to x). The set of operation sequences consists of all sequences in which every read operation returns the value of the latest preceding write operation (the usual read/write semantics). The interface to the mcs consists of calls (also called invocations) and responses on particular objects.

The run-time system takes as input the application program and executes instructions on the mcs. It consists of a process at each node. An operation is a specific instance of an execution of an instruction. A memory operation consists of two parts, a call (to the mcs) and a matching response (from the mcs). 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 system must keep track of the local state of the application process in order to do the evaluation.

An event is a call, a response, or a control operation (condition evaluation). A run (of the run-time 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.

\footnote{The run-time system and mcs may also communicate concerning issues such as rollback and compensating operations, necessary to implement certain optimizations. This communication can also be modeled with calls and responses.}
consistency, becomes a correct solution under hybrid consistency as well, by adding one strong write (to a location that is never read from) to the entry section and by labeling any write in the entry and exit sections as strong, any read as weak and all other instructions as weak. It is a common paradigm in concurrent programming to guarantee that access to shared data is done only within critical sections. Our result yields a very simple and efficient method for executing programs that adhere to this paradigm.

Our second static technique for programming with hybrid consistency is symmetric to the first: We show that, under certain assumptions, programs in which all reads are strong run on hybrid consistent shared memory implementations as if they were sequentially consistent.

The second general approach for programming with hybrid consistency is to run data-race-free programs. (This is analogous to the work of [2, 3, 21, 20].) A data race occurs when two accesses to the same location occur, at least one is a write, and there is no synchronization between them. Data races in a program are considered bad practice: They add to the uncertainty of concurrent programs, beyond what is already implied by the fact that different processes may run at different rates and memory accesses may have variable duration. (Some debuggers even regard data races as bugs in the program.) Methods have been developed to detect and report data races, also called access anomalies; e.g., [5, 12, 13, 14, 29, 32, 33]. It is reasonable to assume that data-race-free programs account for a substantial portion of all concurrent programs. We formally prove that data-race-free programs run on hybrid consistent shared memory implementations as if they were sequentially consistent.

Although many parallel programs are expected to be data-race-free, we cannot ignore the drawbacks of these programs. Proving that a program is data-race-free, even for restricted cases, is NP-hard [31]. Also, it is sometimes difficult to find the exact location of the data race in the program [32]. Our static methods provide an alternative and show that it is not necessary to make a program data-race-free in order to guarantee correct operation. These methods are especially well-suited to applications in which reads greatly outnumber writes (or vice versa).

A variant of hybrid consistency that distinguishes between release and acquire operations has been proposed (e.g., [3, 20, 21, 19]). Informally, release and acquire operations are strong operations, but their effect is not symmetric—a release operation orders all weak operations by the same process that precede it, while an acquire operation orders all weak operations by the same process that follow it. This variant is sometimes called release consistency, but we prefer the name asymmetric hybrid consistency because it highlights the main feature of the consistency condition. Obviously, this further classification puts an additional burden on the programmer (or the compiler and run time system). Although recent research [18, 41] suggests that the benefits from this further classification are somewhat limited (when compared to hybrid consistency), in order to demonstrate the flexibility of our framework, and for compatibility with recent trends in the architecture community we show that "asymmetric" data-race-free programs run on asymmetric hybrid consistent memory implementations as if they were sequentially consistent.

The rest of the paper is organized as follows. Section 2 describes our system model. The modified definitions of three common consistency conditions are given in Section 3. The static...
beyond the scope of this paper.) Instead, our goal is to model the run-time system sufficiently abstractly so that any of a large number of specific run-time systems can fit into this framework. Obviously the run-time system cannot do just anything—the optimizations that it performs should be transparent to the application program. The condition we require of a correct run-time system is that there exists a way (after the fact) to take at least some of the operations performed by the run-time system and order them to be consistent with some “sequential execution” of the program. (Some of the operations performed by the run-time system might end up not being used, for instance if they resulted from an incorrect prediction about a branch. These operations can be ignored in determining whether there is a corresponding sequential execution.) We emphasize that the order in which operations appear to execute is what is important, not the order in which they actually execute.

Our framework incorporates rollback and compensating operations in an implicit manner. In particular, we allow the run-time system to communicate with the mcs in order to perform other operations on the data, not requested by the application programs, but necessary for restoring the state of the shared variables due to incorrect predictions, for example. These operations are ignored when the subset of operations consistent with a sequential execution is taken.

Given this framework, we generalize four known consistency conditions—sequential consistency, weak consistency, hybrid consistency, and release consistency.

Our second contribution addresses the issue of writing programs for hybrid consistency and formally proving their correctness. Hybrid consistency is an efficient and expressive consistency condition [9]; it unifies and generalizes several other consistency conditions appearing in the literature [2, 11, 15]. Memory access operations are classified as either strong or weak. A global ordering is imposed on strong operations at different processes, but not much is guaranteed about the ordering of weak operations at different processes, except for what is implied by their interleaving with the strong operations.

Unfortunately, it is more difficult to program memories that support hybrid consistency than to program memories that support sequential consistency, since the guarantees provided by the former are weaker than those provided by the latter. A way to cope with this problem is to develop rules and transformations for executing programs that were written for sequentially consistent memories on hybrid consistent memories. The benefit is that sequentially consistent executions are easier to reason about while hybrid consistency can be implemented more efficiently. We present two different approaches for turning programs written for sequential consistency into programs that work for hybrid consistency. We believe that these techniques are a first step in developing automatic optimization techniques for the compilation and execution of parallel programs.

The first approach we present is based on statically labeling specific accesses as strong, according to their type. We derive two specific techniques from this approach. First, we prove that programs in which all writes are strong run on hybrid consistent shared memory implementations as if they were sequentially consistent. We then show how this result can be used to produce efficient synchronization code based on mutual exclusion. Specifically, we prove that any non-cooperative solution for the mutual exclusion problem written assuming sequential
following two features. (1) The interface at which conditions are specified in our model is between the program and the system. This is the interface used in [27, 25, 9] and the natural one to use for specification to be independent of implementation. (2) The framework allows for arbitrary optimizations by the system, including especially non-sequential execution of memory accesses. The framework is then used to extend four known consistency conditions for non-sequential execution. Our extensions have two pleasing properties: (1) The conditions are defined for all programs, not just programs that satisfy certain conditions. (2) We give a formal yet intuitive treatment of explicit control instructions, which are crucial for expressing the flow of control in a program and in analyzing its correctness on non-sequential implementations.

Our framework assumes a system consisting of a collection of nodes. At each node there is an application program, a memory consistency system (mcs) process, and a run-time system. An illustration of a node is given in Figure 1. An application program contains instructions to access shared memory and conditional branch instructions. The mcs processes at all the nodes collectively implement the shared objects that are manipulated by the application programs. The run-time system executes the shared memory instructions by interacting with the local mcs process; it bases its decisions as to which instructions to execute on the application program at that node. (We abuse the term run-time system and use it to refer to the combination of the functionality of a compiler, which sees the whole program, and a conventional run-time system, which makes decisions dynamically based on partial knowledge.) In our framework, consistency conditions are specified at the interface between the application program and the system.

A straightforward run-time system would simply submit operations to the mcs one at a time in the order specified by the program. In order to achieve various optimizations however, the run-time system might submit operations out of order, might have multiple operations pending at a time, and might anticipate branches (sometimes incorrectly). We are not interested in the specific algorithm used by the run-time system. (That is another very interesting problem,
1 Introduction

1.1 Overview

Intense interest is currently being focused on shared memory multiprocessors in many disciplines of computer science. The parallelism in multiprocessors offers the potential of greatly increased performance, and shared memory is an attractive communication paradigm. Unfortunately, access to shared memory locations is a major bottleneck for the performance of multiprocessors. The high latency of memory operations is due to the inter-processor communication delay, which increases with the number of processors, and the time to execute memory operations locally.

Many computer architecture techniques have been developed and implemented to hide the latency of memory operations by allowing operations to overlap. These techniques include performing memory accesses in parallel, pipelining memory accesses, initiating accesses out of order, and speculative execution\(^1\). Specific instances include, e.g., \([1, 16, 23, 34, 35, 37, 39, 40]\). Since all these techniques deviate from the sequential order of memory accesses specified by the program, we call them “non-sequential”.

Non-sequential execution of memory accesses complicates the user’s model of memory. A consistency condition for shared memory specifies what guarantees are provided about the values returned in the presence of concurrent accesses. A variety of consistency conditions have previously been proposed for shared memory architectures, e.g., \([8, 7, 2, 3, 9, 15, 21, 19, 25, 27, 28]\). In the presence of non-sequential execution, an obvious question is whether they can still be provided efficiently, or, if not, what guarantees are provided by the optimized system. A further question is how to program these optimized systems with new consistency conditions in a safe and effective manner.

In this paper, we present a theoretical foundation that allows programming to exploit non-sequential execution of memory accesses on multiprocessors. Such a foundation provides a common ground and interface between researchers investigating multiprocessor architectures, concurrent programming languages and parallelizing compilers. We extend four known consistency conditions to allow for non-sequential execution of memory accesses. Our conditions are stated as properties of executions, not hardware implementations. We then present and prove correct several techniques for programming on optimized architectures so as to provide the illusion of sequential consistency, which is easier to reason about. For hybrid consistency, the first approach is to label all writes (or all reads) as strong. The second method, again for hybrid consistency, is to avoid data races. An analogous result is shown for asymmetric hybrid consistency, our attempt at a formalization of release consistency.

1.2 Detailed Description

Our first contribution is a framework for defining consistency conditions that is general enough to allow non-sequential execution of memory operations. This framework is novel in combining the

\(^1\)Predict the outcome of future conditional branches and begin memory accesses for the predicted branch.
Abstract

To enhance performance on shared memory multiprocessors, various techniques have been proposed to reduce the latency of memory accesses, including pipelining of accesses, out-of-order execution of accesses, and branch prediction with speculative execution. These optimizations however can complicate the user's model of memory. This paper attacks the problem of simplifying programming on two fronts.

First, a general framework is presented for defining shared memory consistency conditions that allows non-sequential execution of memory accesses. The interface at which conditions are defined is between the program and the system, and is architecture-independent. The framework is used to generalize four known consistency conditions—sequential consistency, hybrid consistency, weak consistency, and release consistency—for non-sequential execution. Thus familiar consistency conditions can be precisely specified even in optimized architectures.

Second, several techniques are described for structuring programs so that a shared memory that provides the weaker (and more efficient) condition of hybrid consistency appears to guarantee the stronger (and more costly) condition of sequential consistency. The benefit is that sequentially consistent executions are easier to reason about. The first and second techniques statically classify accesses based on their type. This approach is extremely simple to use and leads to a general methodology for writing efficient synchronization code. The third technique is to avoid data races in the program. This technique also works on a simple variant of release consistent hardware, with an appropriate change to the definition of data race.

Precise yet short and comprehensible proofs are provided for the correctness of the programming techniques. Such proofs shed light on the reasons these techniques work; we believe that the insight gained can lead to the development of other techniques.
Shared Memory Consistency Conditions for Non-Sequential Execution: Definitions and Programming Strategies

Hagit Attiya†  Soma Chaudhuri‡  Roy Friedman†  Jennifer L. Welch§

February 21, 1994


†Department of Computer Science, The Technion, Haifa 32000, Israel. Email: hagit@cs.technion.ac.il and roy@cs.technion.ac.il. Partially supported by Technion V.P.R. funds from the Argentinian Research Fund, and by the fund for the promotion of research in the Technion.

‡Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. Email: soma@theory.lcs.mit.edu. On leave from the Department of Computer Science, Iowa State University. Supported in part by NSF grant CCR-89-15206, DARPA contracts N00014-89-J-1988 and N00014-92-J-1033, ONR contract N00014-91-J-1046, and a grant from the ISU Graduate College.

§Department of Computer Science, Texas A&M University, College Station, TX 77843-3112. Email: welch@cs.tamu.edu. Much of this work was performed while this author was with the Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175. Supported in part by NSF grant CCR-9010730, an IBM Faculty Development Award, an NSF Presidential Young Investigator Award CCR-9158478, and TAMU Engineering Excellence funds.