# The Complexity of Sequential Consistency

Phillip B. Gibbons
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Ephraim Korach
Technion – Israel Institute
of Technology
Haifa 32000, Israel

## Abstract

*Sequential consistency is the most-widely used correctness condition for multiprocessor memory systems. This paper explores the complexity of deciding whether an execution of a shared-memory multiprocessor is sequentially consistent. We present the first results showing the NP-completeness of this problem, even for short programs or small machines. We also explore possible augmentations to the memory system: a fast decision algorithm is presented for such an augmented shared memory.*

## 1 Introduction

Shared memory multiprocessors typically promise software writers some high-level view of the memory system. High-level correctness conditions such as sequential consistency [Lam79] or linearizability [HW90] present a conceptually simple framework in which to develop software for parallel machines. A fundamental open problem in the design of shared memory multiprocessors is how to support a high-level correctness condition efficiently and cost-effectively in hardware. A number of designs have been proposed/implemented (cf. [GW88, ShS88, BNR89, ACC+90, AH90a, LLG+90, GGH91, ABM92, GGH92]), varying in (1) the concurrency permitted in the implementation, (2) the cost of the implementation, (3) the reliance on compilers to determine opportunities for increased concurrency, and (4) the requirement that programs satisfy certain restrictions. Two important issues arising in this context are:

- Are there efficient implementations of a high-level correctness condition in its full generality? Or must concurrency be restricted so that only a subset of the possible legal executions are permitted?
- Are there efficient algorithms for detecting when a memory system fails to meet its correctness condition?

This paper addresses these two issues for sequential consistency, the most-widely used correctness condition promised to programmers of shared memory multiprocessors. In a sequentially consistent memory, each execution is indistinguishable (by the processors) from an execution of a serial memory in which only one read or write occurs at a time, in an order consistent with the individual program orders at each processor [Lam79, ABM92]. This paper presents the following answers to the two questions:

Are there efficient algorithms for detecting when a memory system fails to be sequentially consistent?

We consider the problem of detecting violations of sequential consistency, based on the program's observations of the shared memory, i.e. the reads and writes occurring at each processor. We show that verifying whether or not a program observes a sequentially consistent shared memory is NP-complete, even for machines with only three processors or programs in which each processor executes just two reads/writes. We also explore the effect of two augmentations to the shared memory system. If each observed read operation is identified with the write operation responsible for the value read, the problem remains NP-complete. However, if a total order on write operations to each address is known as well, we are able to devise a fast detection algorithm.

Are there efficient implementations of sequential consistency in its full generality?

The above results have implications for implementations of sequentially consistent shared memories. The results demonstrate a trade-off between the amount of state information maintained by the memory system and the feasibility of supporting all possible sequentially consistent executions. For example, consider a (powerful) memory system that maintains state information comprised of a complete trace of the reads and writes occurring at each processor thus far, with the address and value for each. Our results show that it is intractable for the memory system to support sequential consistency in full generality, since the memory



0-8186-3200-3/92 $03.00 © 1992 IEEE

system would need to solve an NP-complete problem. Moreover, it is intractable even if there are only three processors in the machine! On the other hand, our results show that a memory system that maintains a total order on write operations to each address, as well as a mapping from each read to the write responsible for the value read, can in principle support all possible sequentially consistent executions (i.e. without solving an NP-complete problem).

A memory system promising sequential consistency may fail to provide it for a number of reasons. High-performance shared memory multiprocessors employ a variety of techniques to improve their memory system performance; these serve to distance the implementation from the sequential consistency abstraction. Subtle design errors can occur in the memory system architecture, or in the supporting compilers, due to the complexity of the design and the difficulty in reasoning about asynchronous, concurrent systems. Second, various hardware components may fail; such failures are more common in multiprocessors due to the sheer number of components in the memory system. Third, certain implementations used in practice provide only an approximation to sequential consistency, as a trade-off for improved performance. For example, the VAX 8800 implements processor consistency [GLL+90] as an approximation to sequential consistency, observing that programs occurring in practice rarely notice the difference. However, differences do exist, and the naive programmer, failing to satisfy some (undefined) condition on how a "typical" program behaves, fails to observe a sequentially consistent memory.

The primary motivation, though, comes from the use of relaxed consistency models such as *release consistency* [GLL+90, GMG91] or *weak ordering* [AH90b], which provide sequential consistency for a well-defined class of programs. As discussed in Section 6, the memory system may fail to provide a sequentially consistent memory if the program contains data-races.

The combinatorial questions that arise in studying the complexity of sequential consistency are interesting due to the asymmetry between reads and writes. Although reminiscent of the serializability problem for database histories (described in the next section), important differences exist; the sequential consistency problem is shown to be NP-complete under conditions for which the serializability problem is in $P$. Sharp thresholds are observed in which the sequential consistency problem becomes polynomial time; polynomial time algorithms are presented in these cases.

The remainder of the paper is organized as follows. In Section 2, we define the *VSC* problem, and present a proof of its NP-completeness by a reduction from serializability. This motivates the study of restricted versions of the problem. Section 3 presents results for restricting the number of reads/writes at a processor. Section 4 presents results for restricting the number of processors. Section 5 describes a fast algorithm for the general problem on an augmented memory system. The importance of our results to relaxed consistency models is discussed in Section 6. Section 7 describes related work. Finally, Section 8 presents conclusions and open questions. Due to page limitations, this version of the paper omits or sketches many of the algorithms and proofs. The details can be found in the full paper [GK92].

## 2 The VSC problem

### 2.1 A combinatorial problem

We have formalized the detection/checking/verification problem for sequential consistency as follows:

**VERIFYING SEQUENTIAL CONSISTENCY**

INSTANCE: Variable set $A$, value set $D$, finite collection of sequences $S_1, \ldots, S_p$, each consisting of a finite set of memory operations of the form "$read(a, d)$" or "$write(a, d)$", where $a \in A, d \in D$.

QUESTION: Is there a sequence $S \in merge(S_1, \ldots, S_p)$ such that for each $read(a, d)$ in $S$, there is a preceding $write(a, d)$ in $S$ with no other $write(a, d')$ between the two?

In other words, given a sequence of reads and writes initiated by each processor, and the values returned to those reads, is there a single sequence $S$, an interleaving of the processor sequences, such that each read operation returns the value of the most recent write operation in $S$ to the same location. The formalization assumes that each address must be written before it is read; generalizations to handle reads of the initial state of memory are straightforward.

Figure 1 depicts a "yes" instance of the Verifying Sequential Consistency (VSC) problem. A legal sequence or *schedule* is $write(a, 0)$, $write(b, 1)$, $read(b, 1)$, $write(a, 1)$, $write(c, 0)$, $read(a, 1)$. On the other hand, Figure 2 depicts a "no" instance of the VSC problem. It is not possible to merge $S_1$ and $S_2$ into a legal sequence. For example, in the schedule $write(a, 0)$, $write(a, 1)$, $write(b, 1)$, $read(b, 1)$, $read(a, 0)$, the operation $write(a, 1)$ is between $write(a, 0)$ and $read(a, 0)$. We say that such a schedule has a "reads-from violation".

$S_1$ : $write(a,0), write(b,1), read(a,1)$
$S_2$ : $read(b,1), write(a,1), write(c,0)$.

Figure 1: A "yes" instance of the VSC problem.

$S_1$ : $write(a,0), write(a,1), write(b,1)$
$S_2$ : $read(b,1), read(a,0)$.

Figure 2: A "no" instance of the VSC problem.

Corresponding to the two possible augmentations to the shared memory discussed above, we define two restricted versions of the VSC problem. First, we consider the case in which for each read operation, it is known precisely which write was responsible for the value read; the interleaving must respect this relation. This we call the VSC-read problem. The function mapping each read to the responsible write is called a read-mapping. Second, we consider the case in which a total order on write operations to an address is known as well. This we call the VSC-conflict problem.

The VSC problem is reminiscent of the serializability problem for database transactions. The most similar variant is that of *view serializability* [Pap86]. In view serializability, we are given a *history*, a total order, $T$, on a set of reads and writes, where each read or write is associated with a particular transaction, and each read or write contains an address but not a value. The task is to determine if there is a total order on the *transactions* that preserves the reads-from mapping of the original history. The view serializability problem is NP-complete [Pap86]. VSC-read generalizes view serializability by permitting solutions in which the accesses for a processor (transaction) are in order but may not be consecutive. For example, the instance in Figure 1 is a "yes" instance for VSC-read, but a "no" instance for view serializability: both $S = S_1S_2$ and $S = S_2S_1$ have reads-from violations. A second essential difference is that the *input* to a view serializability problem, a consistent total order of the reads and writes, is the desired *output* of the VSC problem.

### 2.2 Verifying sequential consistency is NP-complete

The VSC problem and its variants are in NP, since given a schedule, we can test that the schedule is consistent with the processor sequences, and does not have reads-from violations, in linear time in one pass through the schedule, by simulating the reads and writes.

We will show that the VSC-read problem is NP-complete by a reduction from view serializability.

**Theorem 1** *The VSC-read problem is NP-complete.*

**Proof.** Given a history $H$, an instance of a view serializability problem, we construct an instance of the VSC-read problem as follows. Recall that a history defines a reads-from mapping, and reads and writes in the history do not include values. Let $\alpha$ be an address not in $H$. Let $S_i'$ be the sequence of operations in $H$ for transaction $i$, where each write operation in a transaction is assigned a unique value to write, and each read operation is assigned the value of the closest previous write to the same address in $H$. Let $S_i = W(\alpha,i)S_i'R(\alpha,i)$, for all transactions $i$. This construction of the $S_i$'s ensures that the remainder of each $S_i$ must be scheduled consecutively after its first access $W(\alpha,i)$: each of the $S_i'$'s has accesses to the same $\alpha$, but different values $i$ — hence any schedule that interleaves the $S_i'$'s must violate the reads-from mapping for $\alpha$. It follows that the instance constructed is in VSC-read if and only if the original instance, $H$, is view serializable. □

## 3 Restricting the number of accesses

### 3.1 Two accesses each

In this section, we show that the VSC problem remains NP-complete even if each processor has at most two accesses and each location is written to at most twice. A processor is permitted to have two read accesses. In contrast, if a processor is not permitted to have two read accesses, the problem can be solved in polynomial time, as described in Section 3.2.

Interestingly, the results in this section generalize to the further restriction that considers only schedules in which each processor sequence is required to be a consecutive subsequence (as in the view serializability problem).

We do not know how to modify either the proof in Section 2, or the proof in [Pap79] that serializability is NP-complete, to handle this restricted case. We use instead a reduction from 3SAT. Consider a 3SAT instance, $\mathcal{F}$. We use the notation $(v_i, S(v_i))$ to represent either a true literal $(S(v_i) = T)$ or a false literal $(S(v_i) = F)$ in a clause. We need techniques for simulating an OR and an AND, as well as an assignment of variables that remains in effect until the formula is evaluated. We observe that (1) the second access at a processor must wait for the first, (2) a read must wait for the write to occur, and (3) the second write to an address must wait for all reads of the first write (in order to avoid a reads-from violation). Thus assignment

to $v_i$ can be simulated as follows (sequences are listed in columns):

$$\begin{array}{cccc} W(v_i,T) & R(x,1) & W(v_i,F) & R(x,1) \\ & R(v_i,T) & & R(v_i,F), \end{array}$$

where a single write $W(x,1)$ (shown below) occurs only after the satisfiability of $\mathcal{F}$ has been simulated. Then both writes to $v_i$ cannot occur before $W(x,1)$; this ensures that the initial assignment to each $v_i$ must remain in effect until the satisfiability of $\mathcal{F}$ has been simulated.

An OR is simulated by having two writes to the same location of the same value: a read can be scheduled after either write. For each clause, $C_j = (v_p, S(v_p)) \vee (v_q, S(v_q)) \vee (v_r, S(v_r))$, we have four sequences:

$$\begin{array}{cccc} R(v_p,S(v_p)) & R(v_q,S(v_q)) & R(v_r,S(v_r)) & R(d_j,T) \\ W(d_j,T) & W(d_j,T) & W(c_j,T) & W(c_j,T) \end{array}$$

By observations (1) and (2) above, this ensures that $c_j$ is not set to $T$ unless clause $j$ is satisfied by the guessed truth assignment.

Finally, the AND of the clauses can be simulated by a sequence $R(c_1,T), R(c_2,T), \ldots, R(c_m,T),\ W(x,1)$. However, there are $m+1 > 2$ accesses in this sequence. In order to have only two accesses per processor, we apply observations (1) and (3) above to obtain the following:

$$\begin{array}{ccccc} W(x,0) & R(c_1,T) & R(c_2,T) & \ldots & R(c_m,T) \\ W(x,1) & R(x,0) & R(x,0) & \ldots & R(x,0) \end{array}$$

This ensures that $x$ is not set to 1 unless all clauses have been satisfied by the guessed assignment.

**Lemma 2** *Let $\mathcal{F}$ be an instance of a 3SAT problem, and let $\mathcal{V}$ be the instance of the VSC problem constructed as described above. Then $\mathcal{V}$ is in VSC if and only if $\mathcal{F}$ is in 3SAT.*

**Proof.** In the full paper, we show that if $\mathcal{F}$ is satisfiable, we can construct a schedule in which, for each $i$, the first write to $v_i$ scheduled corresponds to the satisfying truth assignment. Conversely, if $S$ is a valid schedule for $\mathcal{V}$, then the first value written to each $v_i$ is the satisfying assignment. We show that any unsatisfied clause corresponds to a cycle in $S$, a contradiction. □

**Theorem 3** *The VSC problem, restricted to instances in which each sequence contains at most two memory operations and each variable occurs in at most two write operations, is NP-complete.*

In the full paper, we also show the following:

**Lemma 4** *The VSC-read problem, restricted to instances in which each sequence contains at most three memory operations, is NP-complete.*

### 3.2 At most one read

In this section, we further restrict each processor to have at most one read. We permit each write to be read many times, and the two writes for a location may write the same value and hence we do not know *a priori* with which of the two writes a read should be paired. We distinguish between a read with only one possible writer (a *definitive* read) and a read with two possible writers (a *possible* read). We observe that even unread writes impose scheduling constraints, so we cannot assume that there are at least as many reads as writes.

We will construct a graph, $H$, containing directed, undirected, and conditional edges representing the constraints on scheduling implied by the processors and the locations. The following observation simplifies the construction by permitting us to have only "write" nodes in $H$.

**Lemma 5** *For any instance $\mathcal{V}$ of the* VSC *problem, there is a valid schedule for $\mathcal{V}$ if and only if there is a valid schedule, $S'$, for $\mathcal{V}$ such that each read in $S'$ appears after the later of the preceding access at the same processor and the write from which it read, and before any subsequent writes.*

We have the following edges in $H$:

**red:** An (undirected) edge $(x, x'; \mathbf{R})$ between the two writes, $x$ and $x'$, to the same location. These trigger conditional edges, as described below.

**orange:** A directed edge $(x, y; \mathbf{O1})$ from a write $x$ to the write $y$ that immediately follows a definitive read of $x$ on the same processor. A directed edge $(x, y; \mathbf{O2})$ from a write $x$ to a write $y$ on the same processor. A directed edge $(x, x'; \mathbf{O3})$ from a write $x$ to a write $x'$ of the same location if a definitive read at $x'$ follows $x$.

**green:** A directed edge $(w, x'; \mathbf{G})$ labeled $x \rightarrow x'$ from a write $w$ to a write $x'$ if a definitive read of $x$ immediately follows $w$ on its processor and $x$ and $x'$ are two writes to the same location, $w \neq x$ and $w \neq x'$. The label on the edge indicates that this edge is present only if the edge in the label is directed as indicated (e.g. $w$ must be before $x'$ in any valid schedule in which $x$ is before $x'$).

**yellow:** A directed edge $(x, w; \mathbf{Y})$ labeled $x \rightarrow x'$ from a write $x$ to a write $w$ if a possible (not definitive) read of $x$ immediately precedes $w$ on its processor and $x$ and $x'$ are two writes of the same value to the same location, $w \neq x$ and $w \neq x'$.

Papadimitriou [Pap79] defines a "polygraph", in which a pair of edges incident to a node are conditional in that exactly one of the two can be deleted. Here we associate each conditional edge with the orientation of an undirected edge, permitting a more general relationship among sets of edges than possible in a polygraph.

The graph $H$ is used to either create a valid schedule for the instance or determine that a valid schedule does not exist. Note that we can not use topological sort to schedule $H$ until we resolve all unoriented and conditional edges.

**Lemma 6** *If there is an all-orange directed cycle then there is no solution.*

**Lemma 7** *There is an orientation of every red edge in $H$ such that no directed cycles remain if and only if the instance has a valid schedule.*

**Proof.** The proof appears in the full paper. The heart of the proof shows that if the constraints implied by the edges in $H$ are satisfied, creating a schedule of the writes, then the reads can be safely scheduled as prescribed by Lemma 5, with no reads-from violations. □

Thus the goal is to judiciously orient each red edge in $H$ without creating cycles. But each red edge fixes green and/or yellow edges, and can create a collection of mixed cycles that cannot all be eliminated regardless of future choices. Algorithm-VSC1R below shows how to maintain the invariant of Lemma 7 while iteratively orienting red edges.

Since each processor has at most two accesses, yet does not have two reads, the possible sequences are $(w, r)$, $(w, w)$, and $(r, w)$. These are represented in $H$ by one node (denoted type I), two nodes (each denoted type II), and one node (type III), respectively. By examining the conditions under which the various edges in $H$ are defined, we observe the following:

**Lemma 8** *Any directed cycle that contains a green edge must be an all-green cycle. Moreover, the all-green cycles are vertex disjoint.*

### 3.2.1 Algorithm-VSC1R

1. Construct the graph $H$ as described above. If there is an all-orange cycle or an unsatisfied read, output "no solution" and halt. Else we can show that the tail of any **O3** edge is a source node. So delete all **O3** edges in $H$, and continue.

2. Repeat as long as there is a directed all-green cycle in $H$:

   If there is a directed cycle with all fixed green edges then output "no solution". Else if there is a directed cycle with one non-fixed green edge, $e = (x, y; \mathbf{G})$, then delete $e$ and make $y$ a source by orienting the red edge from $y$ to $y'$. Delete $y$. Else, apply the above to any non-fixed green edge, $e = (x, y; \mathbf{G})$, in a directed all-green cycle. Although this is not a "forced" move, it follows by Lemma 6 that this move creates at most one directed cycle with less than two non-fixed green edges.

3. Repeat as long as there is a green edge in $H$: Let $e = (x, y; \mathbf{G})$ be a green edge such that there are no directed edges into $x$. Orient the red edge $(x, x'; \mathbf{R})$, if any, from $x$ to $x'$ and delete $x$.

4. Repeat as long as there is a type (I) source vertex, $x$, in $H$ with an unoriented red edge incident to it: Orient the red edge out of $x$ and delete $x$.

5. By now, we are left with only type (II) and (III) nodes in $H$, and no green edges. Moreover, all remaining red edges are unoriented. Let $S$ be the set of all type (III) vertices in $H$. In this case we can show that there are no edges directed out of $S$. Direct all the red edges between $S$ and $H - S$ into $S$. This implies that any directed cycle is now comprised of either entirely type (II) nodes or entirely type (III) nodes. Remove all the edges between $S$ and $H - S$.

6. If every node in $S$ has a positive (directed) indegree, then there is no solution, since we cannot break all the cycles by orienting red edges. This is because there are no green edges remaining, and for every yellow edge entering a vertex $x$, there is a companion yellow edge also entering $x$, with the same triggering red edge but triggered by an opposite orientation. Thus either orientation of the triggering red edges fixes one of the yellow edges, and the indegree of $x$ remains positive.

   Otherwise, repeat until $S$ is empty: Select a source vertex, $y$, in $H$, i.e. a node with no incoming yellow or orange edges, orient the red edge incident to $y$, if any, out of $y$, and delete $y$.

7. At this point all that remains in $H$ are type (II) nodes, **O2** edges, and unoriented red edges. Repeat until $H$ is empty: Since there are no all-orange cycles, there must be a node, $y$, with no incoming directed edge. Orient the incident red edge, if any, out of $y$, and delete $y$.

8. At this point, we have oriented every red edge in the original $H$, and hence resolved every conditional edge, without creating any directed cycles

in $H$. Topologically sort the nodes in the (now) directed $H$. This creates a schedule of all the writes. Apply Lemma 5 to schedule the reads.

**Theorem 9** *In polynomial time, Algorithm-VSC1R produces a schedule if one exists or else states that there is no solution.*

## 4 Restricting the number of processors

Many multiprocessors have only a small number of processors, e.g. 8, 16, or 32. We have shown that the VSC problem with $O(n)$ processors is NP-complete; in this section, we show that the VSC problem with just three processors is still NP-complete. This contrasts with the serializability problem, which is in $P$ if the number of transactions is restricted to a constant: with $k$ "processors", there are only $k!$ possible serializations to check, regardless of the number of accesses in each transaction. In addition, we present a polynomial time algorithm for the VSC-read problem restricted to two processors.

### 4.1 Three processors

The NP-completeness proof of Section 3 uses a disjoint set of processors for each variable. The difficulty in proving an NP-completeness result for a fixed number of processors is that we do not have as much freedom to schedule operations in an arbitrary order, since the accesses at a processor are totally ordered. Since processors are a scarce resource, care must be taken to ensure that a processor with a read operation is not "idle" (i.e. prevented from scheduling this read) for a long period of time.

Our reduction is from POSITIVE ONE-IN-THREE 3SAT, a variant of 3SAT in which no clause contains a negated literal and we seek a truth assignment such that each clause has exactly one true literal (and hence two false literals). This problem is known to be NP-complete [GJ79]. We construct the instance of the VSC problem, using three processors, depicted in Figure 3.

The idea behind this construction is that the desired truth assignment is the second scheduled write to each $v_i$. Any valid schedule proceeds in stages, enforced by the seven access construction marked (*). For each clause, each of the three processors is satisfied by a particular one-in-three assignment. The subtle part of the construction are the writes marked (**). For any of the three ways to satisfy this clause, this construction frees up the other two processors (by negating variables), yet returns all variables to their original setting (for the next clause). Conversely, for any assignment that does not satisfy this clause, there is no valid scheduling of the accesses in this stage.

| $P1$ | $P2$ | $P3$ | |
|---|---|---|---|
| $W(v_1, T)$ | $W(v_1, F)$ | | |
| ... | ... | | |
| $W(v_n, T)$ | $W(v_n, F)$ | | |
| $W(z, 1)$ | $W(z, 2)$ | $R(z, 1)$ | $(*)$ |
| | | $R(z, 2)$ | $(*)$ |
| $R(z, 3)$ | $R(z, 3)$ | $W(z, 3)$ | $(*)$ |
| $R(v_{p_1}, T)$ | $R(v_{q_1}, T)$ | $R(v_{r_1}, T)$ | $(C_1)$ |
| $R(v_{q_1}, F)$ | $R(v_{r_1}, F)$ | $R(v_{p_1}, F)$ | $(C_1)$ |
| $R(v_{r_1}, F)$ | $R(v_{p_1}, F)$ | $R(v_{q_1}, F)$ | $(C_1)$ |
| $W(v_{p_1}, F)$ | $W(v_{q_1}, F)$ | $W(v_{r_1}, F)$ | $(**)$ |
| $W(v_{q_1}, T)$ | $W(v_{r_1}, T)$ | $W(v_{p_1}, T)$ | $(**)$ |
| | ... | | |
| $W(z, 3m-2)$ | $W(z, 3m-1)$ | $R(z, 3m-2)$ | $(*)$ |
| | | $R(z, 3m-1)$ | $(*)$ |
| $R(z, 3m)$ | $R(z, 3m)$ | $W(z, 3m)$ | $(*)$ |
| $R(v_{p_m}, T)$ | $R(v_{q_m}, T)$ | $R(v_{r_m}, T)$ | $(C_m)$ |
| $R(v_{q_m}, F)$ | $R(v_{r_m}, F)$ | $R(v_{p_m}, F)$ | $(C_m)$ |
| $R(v_{r_m}, F)$ | $R(v_{p_m}, F)$ | $R(v_{q_m}, F)$ | $(C_m)$ |
| $W(v_{p_m}, F)$ | $W(v_{q_m}, F)$ | $W(v_{r_m}, F)$ | $(**)$ |
| $W(v_{q_m}, T)$ | $W(v_{r_m}, T)$ | $W(v_{p_m}, T)$ | $(**)$ |

Figure 3: Transforming an instance of POSITIVE ONE-IN-THREE 3-SAT to an instance of VSC. There are $n$ variables, $v_1, \ldots, v_n$, and $m$ clauses, $C_1, \ldots, C_m$, where $C_i = (v_{p_i}, T) \vee (v_{q_i}, T) \vee (v_{r_i}, T)$, for $p_i$, $q_i$, and $r_i \in \{1, 2, \ldots, n\}$.

**Theorem 10** *The VSC problem restricted to three processors is NP-complete.*

**Proof.** In the full paper, we argue formally the claims in the previous paragraph. The result follows. □

### 4.2 Two processors

In the VSC-read problem, each read is mapped to the write responsible for the value read. Define a cluster to be a write, $w$, and all the reads mapped to $w$. To solve the VSC-read problem for two processors, we begin by constructing a graph, $H$, capturing all scheduling constraints that can be represented as a partial order on the accesses (details are in the full paper). For example, if there are two accesses, $u$ and $v$, to the same location, but in different clusters, on one processor in that order, then we add a directed edge from every access in the cluster for $u$ to every access in the cluster for $v$. If the resulting $H$ has a cycle, there is no solution. However, if $H$ is acyclic, then not all topological sorts of its nodes produce valid schedules, since

not all scheduling constraints can be captured by a partial order. Nevertheless, in the full paper, we describe a simple greedy schedule that does suffice, yielding the following theorem:

**Theorem 11** *There is a polynomial time algorithm for the VSC-read problem restricted to two processor sequences.*

## 5 A fast algorithm for verifying sequential consistency in an augmented memory

In this section, we consider the VSC-conflict problem, in which the shared memory system is augmented to record not only a mapping from each read to the write responsible, as in the VSC-read problem, but also a total order on the write operations to each address. Note that the reads to an address need not be ordered.

**Theorem 12** *There is a simple polynomial time algorithm for the VSC-conflict problem.*

**Proof.** We have not specified the input format for the VSC-conflict problem. Suppose the input is $p$ sequences of four-tuples, $(W/R, a, d, \tau)$, where $\tau = i$ if either the access is the $i^{\text{th}}$ write to $a$ or a read of the $i^{\text{th}}$ write to $a$. Then the VSC-conflict problem can be solved using the following $O(n \log n)$ time algorithm:

1. Sort the tuples in the instance by address field (first) and $\tau$ field, favoring writes ahead of reads. In one pass through the sorted tuples, verify, for each $\tau$ value for an address, that there exists exactly one write, $(W, a, d, \tau)$, with this $\tau$ value and address, and that all reads, $(R, a, d', \tau)$, with this $\tau$ value and address have matching data value $d$, i.e. $d' = d$.

2. Construct a graph, $G$, with a vertex for each read or write. Add an edge between a vertex $u$ and a vertex $v$ if $u$ immediately precedes $v$ in some sequence. Add edges from each write $(W, a, d, \tau)$ vertex to all the read $(R, a, d, \tau)$ vertices and to the next write $(W, a, d', \tau+1)$ vertex, if any. Then the directed graph $G$ has a cycle if and only if we have a "yes" instance.

Other input formats have similar complexity. □

The VSC-conflict problem is motivated by memory systems from which both a read-mapping and a total order on the write operations to each address can be extracted, in principle. For example, the Stanford DASH machine [LLG+90] uses an ownership-based, invalidation-based cache policy, in which multiple processors can have cached copies of a shared memory location — as long as no processor is writing the location — but the protocol ensures that only a single copy of a location exists before a write to that location can update the value. Thus the reads to a location by different processors are not ordered, but the writes to a location are ordered, as well as the writes with respect to the reads. Thus, a counter could be maintained for each address: the counter is incremented on each write operation to that address, stored in the cache line, and passed from owner to owner. Each processor could log the counter value of the address with each of its reads and writes. We suspect, however, that the cost of supporting this VSC-conflict framework is unacceptable in practice.

## 6 Relaxed consistency models

Multiprocessors implementing a relaxed consistency model exploit the fact that programmers typically use synchronization primitives such as locks to prevent data-races among the ordinary accesses. (A *data-race*, or *access anomaly*, occurs when two or more processors access the same location, with at least one writing, without intervening synchronization.) Such multiprocessors provide a sequentially consistent memory as long as the program is data-race-free. Examples include the Stanford DASH machine [LLG+90] and the Tera Computer [ACC+90].

Programmers view the memory system as being sequentially consistent and requiring data-race-free programs, and write their programs accordingly; the details of any particular relaxed consistency model are hidden from them. Even when debugging, it is desirable for the programmer to view the memory system as sequentially consistent, if possible.

Unfortunately, determining whether a program has data-races is undecidable [Ber66], and determining whether an execution (could have) had data-races is still NP-hard even under restrictive assumptions on the program [NM90]. Thus programmers assume the responsibility of ensuring that there are no data-races, often a difficult task. A program that mistakenly contains a data-race may not observe a sequentially consistent memory.

Some programs contain data-races by design. A program may access certain data without the overhead of protective synchronization, if the likelihood of a problem arising is considered to be small. For example, a randomized algorithm may produce a data-race only with small probability. For the high-probability case, the memory system is sequentially consistent, but each

execution is suspect.

For these reasons, a memory system that correctly implements a relaxed consistency model may still fail to provide a sequentially consistent memory to an executing program. On the other hand, regardless of whether or not there are data-races, a memory system that provides a sequentially consistent memory to an executing program satisfies its requirements, and permits the programmer to use sequential consistency in reasoning about the shared memory. Therefore, determining the complexity of sequential consistency is the relevant question in practice, even for relaxed consistency models.

## 7 Related work

To our knowledge there has been no previous work specifically on the VSC problem or its variants. In this section, we outline previous work done on some related topics.

Gharachorloo and Gibbons [GG91] describe an implementation of a release consistent memory that correctly signals either that a sequentially consistent memory was provided or that the program contained a data-race. The implementation exploits the fact that the cache coherence traffic serves to notify a processor whenever another processor attempts a conflicting operation on a location the former processor has cached. Gharachorloo and Gibbons show that although accurate information is provided, both predicates — the program has data-races, the memory was sequentially consistent — are only approximated. Gharachorloo, Gupta, and Hennessy [GGH91] present an aggressive implementation of sequential consistency that speculatively executes read operations, monitors for possible violations of sequential consistency, and rolls back the execution as appropriate if a possible violation is detected. Both these techniques address special cases, in obtrusive manners, of the general scenario considered in this paper.

Because detecting data-races in parallel programs is a difficult task, there has been considerable research effort to develop tools for assisting the programmer in this task (e.g. [HKMC90, AHMN91, NM91]). These papers attempt to overcome the inherent intractability of data-race detection by considering more restrictive classes of programs and conservative/approximate detection of data-races.

Shasha and Snir [ShS88] study compile-time algorithms for weakening the ordering constraints between accesses in a program while still obtaining sequentially consistent executions. The VSC problem arises if the compiler analysis is suspect.

## 8 Conclusions

This paper explores the complexity of deciding whether an execution of a shared-memory multiprocessor is sequentially consistent. We define a combinatorial problem, the VSC problem, which is shown to be NP-complete, even for machines with only three processors or programs in which each processor executes just two shared memory accesses. In addition, we have explored the power of simple augmentations to the memory system. Our results show that it does not suffice to tag each read with the identity of the write responsible for the value it read: the problem is still NP-complete. However, if we can also extract a total order on the writes to each address that must be respected by the schedule, then a fast algorithm exists.

These results demonstrate the difficulty in (1) detecting when an execution of a memory system fails to be sequentially consistent, and (2) supporting all possible sequentially consistent executions in hardware.

The combinatorial problems arising from the properties of reads and writes in an asynchronous shared memory are interesting and largely unexplored. For instance, this area motivates a further study of conditional graphs such as polygraphs or the ones defined in this paper. Moreover, it would be interesting to close some of the P/NP-complete gaps remaining for the VSC problem. For instance, is there a polynomial time algorithm for the VSC problem for two processors? Recently, we have extended the result of Section 4.2 by devising a polynomial time algorithm for the VSC-read problem with any fixed number of processors.

Finally, we have recently analyzed the complexity of verifying linearizability [HW90], another well-known correctness condition for shared memories. Our results show that although the general problem is NP-complete, there is a polynomial time algorithm if the number of processors is fixed or if a read-mapping is provided.


### Acknowledgements

This research was performed while the second author was visiting AT&T Bell Laboratories, Murray Hill, NJ, and supported in part by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS), Rutgers University, Piscataway, NJ. The second author's current affiliation is Ben-Gurion University of the Negev, Beer-Sheva, Israel.

The authors thank Michael Merritt for discussions related to this work.

## References


[ABM92] Y. Afek, G. M. Brown, and M. Merritt. Lazy caching. *ACM Trans. on Programming Languages and Systems*, October 1992. To appear.

[ACC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proc. 1990 International Conf. on Supercomputing*, pages 1–6, November 1990.

[AH90a] S. V. Adve and M. D. Hill. Implementing sequential consistency in cache-based systems. In *Proc. 1990 International Conf. on Parallel Processing*, pages I:47–50, August 1990.

[AH90b] S. V. Adve and M. D. Hill. Weak ordering — a new definition. In *Proc. 17th International Symp. on Computer Architecture*, pages 2–14, May 1990.

[AHMN91] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *Proc. 18th International Symp. on Computer Architecture*, pages 234–243, May 1991.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on Electronic Computers*, EC-15(5):757–763, 1966.

[BNR89] R. Bisiani, A. Nowatzyk, and M. Ravishankar. Coherent shared memory on a distributed memory machine. In *Proc. 1989 International Conf. on Parallel Processing*, pages I:133–141, August 1989.

[GG91] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 316–326, July 1991.

[GGH91] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proc. 1991 International Conf. on Parallel Processing*, pages I:355–364, August 1991.

[GGH92] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *Proc. 19th International Symp. on Computer Architecture*, pages 22–33, May 1992.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[GK92] P. B. Gibbons and E. Korach. The complexity of verifying sequential consistency. Technical report, AT&T Bell Laboratories, Murray Hill NJ, May 1992.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th International Symp. on Computer Architecture*, pages 15–26, May 1990.

[GMG91] P. B. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[GW88] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proc. 15th International Symp. on Computer Architecture*, pages 422–431, June 1988.

[HKMC90] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Proc. 1990 International Conf. on Supercomputing*, pages 74–81, November 1990.

[HW90] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. 17th International Symp. on Computer Architecture*, pages 148–159, May 1990.

[NM90] R. H. B. Netzer and B. P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proc. 1990 International Conf. on Parallel Processing*, pages II:93–97, August 1990.

[NM91] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *Proc. 3rd ACM Symp. on Principles and Practice of Parallel Programming*, pages 133–144, April 1991.

[Pap79] C. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[Pap86] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[ShS88] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *Trans. on Programming Languages and Systems*, 10(2):282–312, 1988.