Testing Shared Memories

Article *in* SIAM Journal on Computing · October 1999

DOI: 10.1137/S0097539794279614 · Source: CiteSeer

2 authors:

Phillip Gibbons
Carnegie Mellon University
366 PUBLICATIONS **27,470** CITATIONS
SEE PROFILE

Ephraim none Korach
Ben-Gurion University of the Negev
76 PUBLICATIONS **1,356** CITATIONS
SEE PROFILE

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 1208–1244, August 1997

014

TESTING SHARED MEMORIES*

PHILLIP B. GIBBONS† AND EPHRAIM KORACH‡

Abstract. Sequential consistency is the most widely used correctness condition for multiprocessor memory systems. This paper studies the problem of testing shared-memory multiprocessors to determine if they are indeed providing a sequentially consistent memory. It presents the first formal study of this problem, which has applications to testing new memory system designs and realizations, providing run-time fault tolerance, and detecting bugs in parallel programs.

A series of results are presented for testing an execution of a shared memory under various scenarios, comparing sequential consistency with linearizability, another well-known correctness condition. Linearizability imposes additional restrictions on the shared memory, beyond that of sequential consistency; these restrictions are shown to be useful in testing such memories.

Key words. sequential consistency, linearizability, multiprocessors, shared memory, testing, NP-completeness

AMS subject classifications. 68M15, 68M07, 68Q60, 68Q22

PII. S0097539794279614

1. Introduction.

Shared-memory multiprocessors typically promise application and system programmers some high-level view of the memory system. High-level correctness conditions such as sequential consistency [26] provide a conceptually simple framework for programming parallel machines. In a *sequentially consistent* memory, each execution is indistinguishable (by the processors) from an execution of a (very fast) serial memory in which only one read or write occurs at a time, in an order consistent with the respective sequences of reads and writes at the individual processors [26, 3]. Sequential consistency is the most widely used correctness condition for multiprocessor memory systems.

A memory system promising sequential consistency may fail to provide it for a number of reasons. First, high-performance shared-memory multiprocessors (cf. [5, 7, 12, 27]) employ a variety of techniques to improve their memory system performance (e.g., buffering, pipelining, caching, multiple paths to memory, parallel access to memory banks); 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 parallel machines due to the multitude of components devoted to providing the large shared-memory system. Third, certain implementations used in practice provide only an approximation to sequential consistency, e.g., *processor consistency*, as a tradeoff for improved performance [16]. Fourth, shared-memory multiprocessors may support *release consistency* [16, 21], which provides a sequentially consistent memory (only) for programs that are free of data races. (A *data race* occurs when two or more processors access the same location, with at least one writing, without intervening synchronization.) The memory system may fail to provide sequential consistency if the program contains data races.

* Received by the editors December 20, 1994; accepted for publication (in revised form) September 15, 1995.
http://www.siam.org/journals/sicomp/26-4/27961.html
† Bell Laboratories, Lucent Technologies, Room 2D-148, 600 Mountain Avenue, Murray Hill, NJ 07974 (gibbons@research.bell-labs.com).
‡ Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel (korach@bgumail.bgu.ac.il). Part of this author's work was done while visiting Bell Laboratories, Murray Hill, NJ.

In this paper, we study the problem of testing shared-memory multiprocessors to determine if they are indeed providing a sequentially consistent memory. We focus on the basic problem of testing whether the memory system provided sequential consistency for a given execution of a parallel program. For the given execution, the test provides certification of the consistency or inconsistency of the memory system during that execution, providing useful feedback whenever the memory system is suspect or the program may contain data races, as discussed above. The test can also be used as a building block in testing new memory system designs and realizations by verifying each execution in a suite of test executions.

This paper provides a formal and systematic study of the complexity of testing the correctness of an execution of a shared memory based on the reads and writes observed by the individual processors. We define the problem *verifying sequential consistency of shared-memory executions* (VSC) and prove that it is an NP-complete problem. This motivates the study of various restrictions on the problem to further characterize its complexity. Some of the variants that we consider are also motivated by their potential utility in testing executions of existing parallel machines.

We compare results obtained for testing for sequential consistency with results obtained for testing for linearizability, another well-known correctness condition. In a *linearizable* shared memory, each execution is indistinguishable from an execution of a serial memory, in which each read or write occurs at a distinct point in time between when it is issued by the processor and when the system acknowledges its completion [22]. We define the problem *verifying linearizability of shared-memory executions* (VL) and show that the additional restrictions imposed by linearizability beyond that of sequential consistency are quite useful in testing such memories. In particular, we present $O(n \log n)$-time algorithms for several variants of the VL problem whose corresponding VSC variants are NP-complete.

In an independent work, Wing and Gong [34] defined and studied the problem of testing and verifying linearizability for arbitrary shared data structures (e.g., a FIFO queue with push and pop operations). They developed a simulation environment for testing implementations (written using a C-Threads package) of shared data structures, using as a building block a procedure for verifying individual executions. This procedure uses a greedy algorithm that runs in exponential time; for this reason, Wing and Gong suggested testing implementations by verifying many, short executions (at most several hundred operations each). They also point out that the general problem they consider is NP-complete. Finally, they presented specifications, implementations, and proofs of correctness (i.e., linearizability) for a number of shared data structures. In contrast to their work, we focus specifically on shared memories, consider both sequential consistency and linearizability, and present fast algorithms for several important variants of these testing problems.

Other related previous work includes work on devising a suite of simple programs to help test whether the memory system is providing sequential consistency or a weaker correctness condition [11], work on detecting violations of sequential consistency within the memory system itself [14, 15], work on testing the serializability of database transactions [31], work on detecting data races (e.g., [2, 23, 28, 29]), work on proving that weak memory systems provide sequential consistency for programs that are free of data races (e.g., [1, 20, 21]), work on testing uniprocessor memories [9], work on algorithms for testing data structures on uniprocessors (e.g., [10]), work on verifying specific properties of cache-coherence protocols (e.g., [32] and the references therein), work on computing with faulty shared memories [4], work on de-

termining minimal ordering constraints needed to preserve sequential consistency [33], and work on comparing implementations of sequential consistency versus linearizability (e.g., [8]). However, none of this work addresses the general testing questions considered in this paper.

1.1. The testing problems. During an execution of a parallel program on a given multiprocessor, processors request to read or write particular shared-memory locations as dictated by the program, and the memory system responds to each request with a return value or acknowledgment. Associated with each processor is a total order on its shared-memory operations, denoted its *program order*. Associated with each read operation, $read(a, d, t_1, t_2)$, issued by a processor are the address a of the shared-memory location read, the value d returned for the read, the time t_1 the read was issued, and the time t_2 of the response. Likewise, associated with each write operation, $write(a, d, t_1, t_2)$, issued by a processor are the address a of the shared-memory location written, the value d written, the time t_1 the write was issued, and the time t_2 of the response. The times t_1 and t_2 define an interval of time for an operation: t_1 is the *start-of-interval* time for the operation and t_2 is the *end-of-interval* time for the operation.

We consider testing procedures in which for each processor we are given its sequence of shared-memory operations. Our goal is to determine whether or not the sequences can be interleaved as required by the correctness condition. For example, both sequential consistency and linearizability require that in the interleaved sequence, each read operation returned the value that was written by the last preceding write to the same location (the usual read/write semantics).

Sequential consistency. Sequential consistency is based on respecting the program orders and the read/write semantics, while ignoring the start-of-interval and end-of-interval times. We define the *verifying sequential consistency of shared-memory executions* (VSC) problem as follows.

VERIFYING SEQUENTIAL CONSISTENCY OF SHARED-MEMORY EXECUTIONS

INSTANCE: Variable set A, value set D, finite collection of nonempty 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, an interleaving of $S_1, \ldots, S_p$, 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?

We denote such a sequence S as a *legal schedule* for the instance; such a schedule certifies that the memory system provided sequential consistency for the execution corresponding to the instance. If there is no legal schedule, then the memory system failed to provide sequential consistency.

Figure 1 depicts a positive instance of the VSC problem. Figure 2 depicts a negative instance.

Linearizability. In contrast, linearizability adds the further constraint that the schedule S must respect the time intervals for the operations. We define the *verifying linearizability of shared-memory executions* (VL) problem as follows.

VERIFYING LINEARIZABILITY OF SHARED-MEMORY EXECUTIONS

INSTANCE: Variable set A, value set D, finite collection of nonempty sequences $S_1, \ldots, S_p$, each consisting of a finite set of memory operations of the form "$read(a, d, t_1, t_2)$" or "$write(a, d, t_1, t_2)$," where $a \in A$, $d \in D$, and t_1 and t_2 are positive rationals, $t_1 < t_2$, defining

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

FIG. 1. *A positive instance of the VSC problem. In fact, there are two different legal schedules:* $write(a,0)$, $write(b,1)$, $read(b,1)$, $write(a,1)$, $read(a,1)$, $write(c,0)$ *and* $write(a,0)$, $write(b,1)$, $read(b,1)$, $write(a,1)$, $write(c,0)$, $read(a,1)$.

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

FIG. 2. *A negative instance of the VSC problem. It is not possible to merge* S_1 *and* S_2 *into a legal schedule. 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* $read(a,0)$. *Although* $write(a,1)$ *is an unread write, its existence makes this a negative instance.*

an interval of time such that all intervals in an individual sequence are pairwise disjoint, and t_1 and t_2 are unique rationals in the overall instance.

QUESTION: Is there an assignment of a distinct time to each operation such that

1. each time is within the interval associated with the operation;
2. for each $read(a,d,\tau_1,\tau_2)$, there is a $write(a,d,t_1,t_2)$ assigned an earlier time, with no other $write(a,d',t_1',t_2')$ assigned a time between the two?

Such a time assignment defines a *legal schedule* S that totally orders the operations in the instance. The memory system provided linearizability for the execution corresponding to the instance if and only if there is a legal schedule.

As in the VSC problem, a legal schedule is an interleaving of the individual processor sequences. Thus any linearizable execution is also sequentially consistent. However, as shown below, not all sequentially consistent executions are linearizable.

Figure 3 depicts a positive instance of the VL problem. Thus if the start-of-interval and end-of-interval times are removed from the instance, it is necessarily a positive instance of the VSC problem. Figure 4 depicts a negative instance of the VL problem; this particular instance corresponds to a positive instance of the VSC problem.

For both the VSC and VL problems, the formalization assumes that each variable (i.e., each shared-memory location) must be written before it is read; generalizations to handle reads of the initial state of memory are straightforward. A schedule has a *reads-from violation* if it has a read operation such that either there is no preceding write operation with the same address and value or there is an intervening write operation with the same address but a different value; such a schedule is not legal.

1.2. Results in this paper. Table 1 highlights and compares our main results for the VSC and VL problems. Both problems are NP-complete, as indicated in the table. Three restricted versions of the two problems are studied: bounding the number of operations in each processor sequence, bounding the number of locations in the instance, and bounding the number of processors. A "w.l.o.g." in Table 1

$$S_1: \quad write(a,1,1,3), write(b,1,4,6), write(c,1,7,8).$$
$$S_2: \quad read(b,1,2,5), read(a,1,9,10).$$

FIG. 3. *A positive instance of the VL problem. There are many possible legal time assignments, each of which schedules the write of a, then the write of b, then the read of b, then the write of c, and finally the read of a. If the start-of-interval and end-of-interval times are removed from the instance, we have a positive instance of the VSC problem.*

$$S_1: \quad write(a,0,1,2), read(a,0,5,6).$$
$$S_2: \quad write(a,1,3,4).$$

FIG. 4. *A negative instance of the VL problem. Since the write of value* 1 *must be assigned a time between the write of value* 0 *and the read of value* 0*, there is no legal VL schedule. However, the schedule* $write(a,0)$*,* $read(a,0)$*,* $write(a,1)$ *is a legal VSC schedule.*

indicates that the complexity of the problem is not affected by the given restriction. In addition, we define and study four important variants on the VSC and VL problems; these variants differ in the additional information provided as input, as detailed in section 4. Perhaps somewhat surprisingly, the VSC problem is NP-complete for all but one of the variants considered, in contrast with the results obtained for the VL problem. We also show how our algorithmic results can be extended to handle atomic read–modify–write operations, with no asymptotic penalty.

Our algorithms have small constants and hence are suitable for testing real shared memories (see [17, 19] for implementation details).

Since this is the first paper to study these problems systematically, a number of our NP-completeness results are obtained using somewhat standard techniques. For more interesting constructions, we refer the reader particularly to Theorems 4.3, 4.11, 3.5, and 2.7 of this paper.

The combinatorial questions that arise in studying these testing problems are interesting due to the asymmetry between reads and writes, and the fact that the constraints on legal schedules imposed by reads and writes to the same location cannot be represented as a partial order.

Outline of the paper. The remainder of this paper is organized as follows. Section 2 presents our results for the VSC problem, showing the problem is NP-complete, even with only two operations per processor, only two locations, or only three processors. These results are contrasted with results for the serializability problem for database transactions. Section 3 presents our results for the VL problem, showing that the problem is NP-complete even with only one operation per processor and only one location and, on the other hand, presenting an $O(n \log n)$-time algorithm when the number of processors is fixed. Section 4 presents our results for four variants of the VSC and VL problems that provide additional information as input, specifically, a read-mapping, write-order, read&write only, and conflict-order. It also presents our extensions to handle atomic read-modify-write operations.

Preliminary versions of this work appeared in [18, 19].

2. Verifying sequential consistency. We begin this section with a proof that the VSC problem is NP-complete, even with only two operations per processor (section 2.1). Then the VSC problem is shown to be NP-complete with only two locations

TABLE 1
A summary of the main results of this paper.

variant	*VSC result*	*VL result*
general problem	NP-complete	NP-complete
2 operations per proc	NP-complete	w.l.o.g.
2 locations	NP-complete	w.l.o.g.
3 processors	NP-complete	$O(n \log n)$
read-mapping	NP-complete	$O(n \log n)$
write-order	NP-complete	$O(n \log n)$
read&write only	NP-complete	NP-complete
conflict-order	$O(n \log n)$	$O(n \log n)$

(section 2.2) or only three processors (section 2.3). We conclude this section with some comparisons between the VSC problem (and the results obtained) and the serializability problem for database transactions (section 2.4).

2.1. The VSC problem is NP-complete. The VSC problem is in NP since given a schedule of the reads/writes in the processor sequences, 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, simulating the operations.

THEOREM 2.1. *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.*

Proof. We use a reduction from the 3-Satisfiability (3SAT) problem [13]. Consider a 3SAT instance $\mathcal{F}$ with n variables, $v_1, \ldots, v_n$, and m clauses, $C_1, \ldots, C_m$. We use the notation $(v_i, S(v_i))$ to represent either the variable v_i (when $S(v_i) = \mathsf{T}$) or its complement $\overline{v}_i$ (when $S(v_i) = \mathsf{F}$) in a clause.

To reduce 3SAT to VSC, techniques are needed 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 in all legal schedules, the following must hold:

1. The second operation in a processor sequence must not precede the first.
2. A read operation must not precede the first write operation with the same address and value.
3. A second write operation to an address, writing a different value than the first such write, must not precede any read operation of the first write (in order to avoid a reads-from violation).

Thus assignment to variable v_i can be simulated using the following four sequences, V_i^1, V_i^2, V_i^3, and V_i^4 (listed in columns):

$$\begin{array}{cccc} \underline{V_i^1} & \underline{V_i^2} & \underline{V_i^3} & \underline{V_i^4} \\ W(v_i, \mathsf{T}) & R(x,1) & W(v_i, \mathsf{F}) & R(x,1) \\ & R(v_i, \mathsf{T}) & & R(v_i, \mathsf{F}) \end{array},$$

where a 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_1^1	V_1^2	V_1^3	V_1^4				
$W(v_1,\mathsf{T})$	$R(x,1)$	$W(v_1,\mathsf{F})$	$R(x,1)$				
	$R(v_1,\mathsf{T})$		$R(v_1,\mathsf{F})$	C_1^1	C_1^2	C_1^3	C_1^4
				$R(v_1,\mathsf{T})$	$R(v_3,\mathsf{F})$	$R(v_4,\mathsf{T})$	$R(d_1,\mathsf{T})$
V_2^1	V_2^2	V_2^3	V_2^4	$W(d_1,\mathsf{T})$	$W(d_1,\mathsf{T})$	$W(c_1,\mathsf{T})$	$W(c_1,\mathsf{T})$
$W(v_2,\mathsf{T})$	$R(x,1)$	$W(v_2,\mathsf{F})$	$R(x,1)$				
	$R(v_2,\mathsf{T})$		$R(v_2,\mathsf{F})$	C_2^1	C_2^2	C_2^3	C_2^4
				$R(v_1,\mathsf{F})$	$R(v_2,\mathsf{F})$	$R(v_4,\mathsf{T})$	$R(d_2,\mathsf{T})$
V_3^1	V_3^2	V_3^3	V_3^4	$W(d_2,\mathsf{T})$	$W(d_2,\mathsf{T})$	$W(c_2,\mathsf{T})$	$W(c_2,\mathsf{T})$
$W(v_3,\mathsf{T})$	$R(x,1)$	$W(v_3,\mathsf{F})$	$R(x,1)$				
	$R(v_3,\mathsf{T})$		$R(v_3,\mathsf{F})$	A^0	A^1	A^2	
				$W(x,0)$	$R(c_1,\mathsf{T})$	$R(c_2,\mathsf{T})$	
V_4^1	V_4^2	V_4^3	V_4^4	$W(x,1)$	$R(x,0)$	$R(x,0)$	
$W(v_4,\mathsf{T})$	$R(x,1)$	$W(v_4,\mathsf{F})$	$R(x,1)$				
	$R(v_4,\mathsf{T})$		$R(v_4,\mathsf{F})$				

FIG. 5. *An example of the construction for transforming a* 3SAT *instance to a VSC instance in which each processor sequence has at most two operations and each address is written at most twice. Shown here are the sequences obtained for the* 3SAT *instance* $(v_1 \vee \overline{v}_3 \vee v_4) \wedge (\overline{v}_1 \vee \overline{v}_2 \vee v_4)$.

$(v_r, S(v_r))$, we have four sequences C_j^1, C_j^2, C_j^3, and C_j^4:

C_j^1	C_j^2	C_j^3	C_j^4
$R(v_p, S(v_p))$	$R(v_q, S(v_q))$	$R(v_r, S(v_r))$	$R(d_j,\mathsf{T})$
$W(d_j,\mathsf{T})$	$W(d_j,\mathsf{T})$	$W(c_j,\mathsf{T})$	$W(c_j,\mathsf{T})$

.

Four sequences, as opposed to three, are used in this construction since we are permitted at most two writes to a location. By observations 1 and 2 above, this ensures that the location c_j is not set to T unless clause C_j is satisfied by the guessed truth assignment.

Finally, the AND of the clauses is simulated by the following $m+1$ sequences, $A^0, A^1, \ldots, A^m$:

A^0	A^1	A^2	$\cdots$	A^m
$W(x,0)$	$R(c_1,\mathsf{T})$	$R(c_2,\mathsf{T})$		$R(c_m,\mathsf{T})$
$W(x,1)$	$R(x,0)$	$R(x,0)$		$R(x,0)$

.

The leftmost sequence ensures that $W(x,1)$ is the "second" write to its address. Thus by observation 3 above, x is not set to 1 unless all clauses have been satisfied by the guessed assignment.

This construction uses $4n+5m+1$ sequences and $n+2m+1$ distinct addresses. An example is given in Figure 5.

LEMMA 2.2. *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 a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable. We will construct a schedule in which, for each i, the first write to v_i scheduled corresponds to the satisfying truth assignment. Let $T(v_1), \ldots, T(v_n)$, where $T(v_i) \in \{\mathsf{T}, \mathsf{F}\}$, be a satisfying assignment for $\mathcal{F}$. We construct the following schedule for $\mathcal{V}$:

1. first, $W(v_1, T(v_1)), \ldots, W(v_n, T(v_n))$;

2. then for $j = 1, 2, \ldots, m$, all sequences C_j^t whose read is $R(v_k, T(v_k))$ for some k, followed by C_j^4 if either C_j^1 or C_j^2 has been scheduled;
3. then $W(x, 0)$ from A^0, followed by the sequences A^1 to A^m, followed by $W(x, 1)$ from A^0;
4. then for $i = 1, 2, \ldots, n$, if $T(v_i) = \mathsf{T}$, the sequences V_i^2, V_i^3, and V_i^4; otherwise, if $T(v_i) = \mathsf{F}$, the sequences V_i^4, V_i^1, and V_i^2;
5. finally, for $j = 1, 2, \ldots, m$, any remaining sequences C_j^1, C_j^2, or C_j^3, followed by C_j^4 if it has yet to be scheduled.

The reader may verify that this is a legal schedule.

Conversely, suppose $\mathcal{V}$ is a positive VSC instance. If S is a legal schedule for $\mathcal{V}$, then the first value written to each v_i will be our satisfying assignment. We show that any unsatisfied clause corresponds to a cycle in S; a contradiction. For $i = 1, 2, \ldots, n$, let $T(v_i) = \mathsf{T}$ if $W(v_i, \mathsf{T})$ is before $W(v_i, \mathsf{F})$ in S; otherwise, let $T(v_i) = \mathsf{F}$. Suppose $T(v_1), \ldots, T(v_n)$ is *not* a satisfying assignment for $\mathcal{F}$, and let

$$C_k = (v_p, S(v_p)) \vee (v_q, S(v_q)) \vee (v_r, S(v_r))$$

be an unsatisfied clause. Let $\neg S(v_p)$, $\neg S(v_q)$, and $\neg S(v_r)$ denote the complement of $S(v_p)$, $S(v_q)$, and $S(v_r)$, respectively. Since C_k is not satisfied, $W(v_p, \neg S(v_p))$ is before $W(v_p, S(v_p))$, $W(v_q, \neg S(v_q))$ is before $W(v_q, S(v_q))$, and $W(v_r, \neg S(v_r))$ is before $W(v_r, S(v_r))$ in S. Since there are no reads-from violations in S and only two writes to v_p, all of the $R(v_p, \neg S(v_p))$ operations are before any of the $R(v_p, S(v_p))$ operations in S; similarly for v_q and v_r. Since S is a legal schedule, some $W(c_k, \mathsf{T})$ precedes the $R(c_k, \mathsf{T})$ in A^k, which precedes $W(x, 1)$, which precedes all $R(x, 1)$, including the one that precedes $R(v_p, \neg S(v_p))$, which as argued above precedes the $R(v_p, S(v_p))$ from C_k^1, which precedes the $W(d_k, \mathsf{T})$ on C_k^1. Likewise for v_q, some $W(c_k, \mathsf{T})$ precedes the $W(d_k, \mathsf{T})$ on C_k^2. One of these $W(d_k, \mathsf{T})$'s must precede the $R(d_k, \mathsf{T})$ on C_k^4 which precedes the $W(c_k, \mathsf{T})$ on C_k^4. A similar argument for v_r shows that some $W(c_k, \mathsf{T})$ also precedes $W(c_k, \mathsf{T})$ on C_k^3. This is a contradiction since only C_k^3 and C_k^4 contain $W(c_k, \mathsf{T})$. □

Since the above transformation can be done in polynomial time, Theorem 2.1 is proved. □

Note that instances with only one memory operation per processor can be solved by simply checking that there exists a write operation with the same address and value as each read operation.

We also observe that instances with long processor sequences can always be transformed to equivalent instances with at most three memory operations per processor.

OBSERVATION 2.3. *There is a linear-time reduction from the VSC problem with p sequences, n operations, and k variables to the VSC problem with n sequences, $O(n)$ operations, and $k + 1$ variables such that each sequence contains at most three operations.*

Proof. Let α be an address not in the original instance. For $i = 1, \ldots, p$, we replace the ith sequence in the instance, $s_1^{(i)}, s_2^{(i)}, \ldots, s_{m_i}^{(i)}$, with the following m_i sequences:

$$\begin{array}{ccccc} s_1^{(i)} & R(\alpha, x_1^{(i)}) & \cdots & R(\alpha, x_{m_i-2}^{(i)}) & R(\alpha, x_{m_i-1}^{(i)}) \\ W(\alpha, x_1^{(i)}) & s_2^{(i)} & & s_{m_i-1}^{(i)} & s_{m_i}^{(i)} \\ & W(\alpha, x_2^{(i)}) & & W(\alpha, x_{m_i-1}^{(i)}) & \end{array},$$

where $\forall i, j, i', j'$, $x_j^{(i)} = x_{j'}^{(i')}$ if and only if $i = i'$ and $j = j'$. There are a total of $\sum_{i=1}^{p}(3m_i - 2) = 3n - 2p$ operations. Since each new data value is unique, the

operations in the ith sequence of the original instance appear in sequence order in any legal schedule of the constructed instance. The reader may verify that this constructed instance is a positive instance if and only if the original instance is a positive instance. □

2.2. The VSC problem with two locations. We show that the VSC problem is NP-complete even when only two locations are used.

THEOREM 2.4. *The VSC problem restricted to instances with only two variables is* NP*-complete.*

Proof. Our reduction from 3SAT is depicted in Figure 6. We use two variables, a and b. Variable a is used to select a truth setting. The writes to a in the first sequence set instance variables to *true*; the writes to a in the second sequence set instance variables to *false*. For each instance variable, the second such write establishes the truth setting. Variable b is used to ensure that exactly one assignment is selected per instance variable by forcing both writes to a for this instance variable to be scheduled before any writes to a for the next instance variable.

The three sequences for a clause C_j begin with two reads, corresponding to a particular literal in the clause. The first read can be scheduled only after the variable's truth setting has been established; the second read can then be scheduled if the assignment to the variable matches the assignment needed for the clause.

The final write in the first sequence together with the first read in the third sequence ensure that all of the first and second sequences must be scheduled before any of the third sequence. The $2n$ writes at the end of the third sequence are used to clean up the remaining reads after the satisfiability of the 3SAT instance has been simulated. The reads in the third sequence demand that for each clause, at least one of the writes is scheduled prior to the cleanup: in order to schedule the $R(b, 2n + j)$ operation, we must first schedule a $W(b, 2n+j)$ operation on behalf of clause C_j. This in turn requires that the two reads to a in some sequence for C_j be scheduled prior to the cleanup, and this is possible only if that particular literal in C_j is satisfied.

LEMMA 2.5. *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 a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable. We will construct a schedule in which, for each v_i, the latter of $W(a, \mathsf{T}_i)$ and $W(a, \mathsf{F}_i)$ scheduled corresponds to the satisfying truth assignment. Let $T(v_1), \ldots, T(v_n)$ be a satisfying assignment for $\mathcal{F}$, where $T(v_i) \in \{\mathsf{T}, \mathsf{F}\}$. We construct the following schedule for $\mathcal{V}$:

1. Repeat the following for $i = 1, 2, \ldots, n$: Consider group i. If $T(v_i) = \mathsf{T}$, schedule $W(a, \mathsf{F}_i)$, then $W(a, \mathsf{T}_i)$. Otherwise, schedule $W(a, \mathsf{T}_i)$, then $W(a, \mathsf{F}_i)$. Next, schedule $W(b, 2i-1)$, $R(b, 2i-1)$, $W(b, 2i)$. Then schedule all $R(b, 2i)$ operations. If $T(v_i) = \mathsf{T}$, schedule all $R(a, \mathsf{T}_i)$ operations. Otherwise, schedule all $R(a, \mathsf{F}_i)$ operations.
2. Schedule $W(a, \alpha)$, $W(b, \beta)$, $R(b, \beta)$.
3. Repeat the following for $p = 1, 2, \ldots, m$: Schedule any $W(b, 2n+p)$ for which both reads above it have been scheduled. Since each clause of $\mathcal{F}$ is satisfied, at least one such write can be scheduled. Then schedule $R(b, 2n + p)$.
4. Finally, we schedule the cleanup writes together with all unscheduled reads in the $3m$ clause sequences. At the very end, schedule all remaining writes to b in these clause sequences.

The reader may verify that this is a legal schedule.

Conversely, suppose $\mathcal{V}$ is a positive instance. If S is a legal schedule for $\mathcal{V}$, then

First, we have the following three sequences:

$W(a, \mathsf{T}_1)$	$W(a, \mathsf{F}_1)$	$R(b, \beta)$
$W(b, 1)$	$R(b, 1)$	$R(b, 2n+1)$
$R(b, 2)$	$W(b, 2)$	$R(b, 2n+2)$
$W(a, \mathsf{T}_2)$	$W(a, \mathsf{F}_2)$	:
$W(b, 3)$	$R(b, 3)$	$R(b, 2n+m)$
$R(b, 4)$	$W(b, 4)$	$W(a, \mathsf{T}_1)$
:	:	$W(a, \mathsf{F}_1)$
$W(a, \mathsf{T}_n)$	$W(a, \mathsf{F}_n)$	$W(a, \mathsf{T}_2)$
$W(b, 2n-1)$	$R(b, 2n-1)$	$W(a, \mathsf{F}_2)$
$R(b, 2n)$	$W(b, 2n)$	:
$W(a, \alpha)$		$W(a, \mathsf{T}_n)$
$W(b, \beta)$		$W(a, \mathsf{F}_n)$

Then for each clause C_j, say $C_j = v_p \vee \overline{v}_q \vee v_r$, we have the following three sequences:

$R(b, 2p)$	$R(b, 2q)$	$R(b, 2r)$
$R(a, \mathsf{T}_p)$	$R(a, \mathsf{F}_q)$	$R(a, \mathsf{T}_r)$
$W(b, 2n+j)$	$W(b, 2n+j)$	$W(b, 2n+j)$

FIG. 6. *Transforming an instance of* 3SAT *to an instance of VSC with just two locations, a and b. There are n variables, $v_1, \ldots, v_n$, and m clauses, $C_1, \ldots, C_m$ in the* 3SAT *instance. We construct a VSC instance with $3m+3$ sequences. $W(a, d)$ designates a write to location a of the value d; $R(a, d)$ designates a read from location a of the value d. Some data values are expressed as arithmetic expressions, e.g., $2n+m$, $2n+j$, $2p$, $2q$, and $2r$, indicating a value that is the result of evaluating the expression using the values of n, m, j, p, q, and r appropriate to the overall instance, the particular clause, or the particular variable.*

let $\mathcal{T}$ be the truth assignment corresponding to, for each v_i, the latter of $W(a, \mathsf{T}_i)$ and $W(a, \mathsf{F}_i)$ (from the first two sequences) scheduled. Let $S = S_1 S_2 S_3$, where S_1 is the prefix of S up to and including $W(b, \beta)$ and S_3 is the suffix of S starting with $W(a, \mathsf{T}_1)$ from the cleanup. All operations in the first two sequences are in S_1, but none of the last sequence is in S_1. Since the value of b is β at the end of S_1 and any writes to b not in S_1 write values in $[2n+1..2n+m]$, all of the reads of b in the clause sequences are in S_1.

Consider a clause C_p and its three sequences. If none of the three reads to a in these sequences are in S_1, then since the value of a is α at the end of S_1, all must be in S_3. But therefore all of the writes $W(b, 2n+p)$ are also in S_3. However, $R(b, 2n+p)$ must be in S_2, a contradiction. Consider a literal v_i in C_p (the case of $\overline{v}_i$ is symmetric). Since any $R(b, 2i)$ must be after the $W(b, 2i)$, which in turn must be after $W(b, 2i-1)$, this implies that $R(a, \mathsf{T}_i)$ must be after both $W(a, \mathsf{T}_i)$ and $W(a, \mathsf{F}_i)$. Therefore, in the three sequences of C_p, at least one of the reads to a in S_1 must read the value written by the second of these two writes. It follows that C_p is satisfied by T. □

Since the above transformation can be done in polynomial time, Theorem 2.4 follows. □

A simple modification of the previous construction shows that the VSC problem is NP-complete even when both the number of locations and the number of operations per sequence are small constants.

COROLLARY 2.6. *The VSC problem restricted to instances with only two variables is* NP*-complete, even if each sequence contains at most three memory operations.*

Proof. Since we are restricted to only two variables, the general reduction of Observation 2.3 cannot be applied. Instead, we divide the first sequence in Figure 6, with its $3n+2$ operations, into $n+1$ sequences, one with the first two operations only (i.e., creating the sequence $W(a, \mathsf{T}_1)$, $W(b, 1)$), and the remaining n with subsequent sets of three operations. We divide the second sequence in Figure 6, with its $3n$ operations, into n sequences of three operations each. Finally, we replace the last sequence in Figure 6, with its $2n + m + 1$ operations, with the following $m + n$ sequences of three operations each: the sequence $R(b, \beta)$, $R(b, 2n + 1)$, $W(a, \alpha_2)$; for $i = 2, \ldots, m - 1$, the sequence $R(a, \alpha_i)$, $R(b, 2n + i)$, $W(a, \alpha_{i+1})$; the sequence $R(a, \alpha_m)$, $R(b, 2n + m)$, $W(b, \beta_2)$; and finally, for $i = 1, \ldots, n$, the sequence $R(b, \beta_2)$, $W(a, \mathsf{T}_i)$, $W(a, \mathsf{F}_i)$. The reader may verify that this new construction is a positive instance if and only if the construction in Figure 6 is a positive instance. Thus the corollary follows from the proof of Theorem 2.4. □

2.3. The VSC problem for three 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.

The previous NP-completeness proofs in this paper use a disjoint set of processors for each clause; some also use a disjoint set of processors for each variable. The difficulty in proving an NP-completeness result for a small fixed number of processors is that we do not have as much freedom to schedule operations in an arbitrary order, since the total order on operations at a processor must be respected by any legal schedule. Since processors are a scarce resource, care must be taken to ensure that the construction permits steady progress through each processor sequence, unless the intent is to construct a negative instance. In particular, for each read operation r, there is a corresponding write operation that can be scheduled closely after the operation preceding r at its processor.

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 [13]. We construct the instance of the VSC problem, using three processors, depicted in Figure 7.

The idea behind this construction is that the desired truth assignment is the second scheduled write to each v_i. Any legal schedule proceeds in stages, enforced by the seven-operation 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 legal scheduling of the operations in this stage.

THEOREM 2.7. *The VSC problem restricted to three processors is* NP*-complete.*

Proof. Let $\mathcal{F}$ be an instance of a POSITIVE ONE-IN-THREE 3SAT problem, and let $\mathcal{V}$ be the instance of the VSC problem constructed as depicted in Figure 7. We will show that $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is a positive instance.

Suppose $\mathcal{F}$ is one-in-three satisfiable. We will construct a schedule in which, for each i, the second write to v_i scheduled corresponds to the satisfying truth assignment.

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

FIG. 7. *Transforming an instance of* POSITIVE ONE-IN-THREE 3SAT *to an instance of VSC. There are* n *variables,* $v_1, \ldots, v_n$, *and* m *clauses,* $C_1, \ldots, C_m$, *where* $C_i = (v_{p_i}, \mathsf{T}) \vee (v_{q_i}, \mathsf{T}) \vee (v_{r_i}, \mathsf{T})$, *for* p_i, q_i, *and* $r_i \in \{1, 2, \ldots, n\}$.

Let $T(v_1), \ldots, T(v_n)$, where $T(v_i) \in \{\mathsf{T}, \mathsf{F}\}$, be a satisfying assignment for $\mathcal{F}$. We construct the following schedule for $\mathcal{V}$:

1. First, for $i = 1, 2, \ldots, n$, the pair $W(v_i, \mathsf{T})$, $W(v_i, \mathsf{F})$ if $T(v_i) = \mathsf{F}$, or the pair $W(v_i, \mathsf{F})$, $W(v_i, \mathsf{T})$ if $T(v_i) = \mathsf{T}$;
2. then $W(z, 1)$, $W(z, 2)$, $R(z, 1)$, $R(z, 2)$, $W(z, 3)$, $R(z, 3)$, $R(z, 3)$.
3. Since clause $C_1 = (v_{p_1}, \mathsf{T}) \vee (v_{q_1}, \mathsf{T}) \vee (v_{r_1}, \mathsf{T})$ is satisfied by a one-in-three assignment, exactly one of the following is true: (1) $T(v_{p_1}) = \mathsf{T}$, $T(v_{q_1}) = \mathsf{F}$, and $T(v_{r_1}) = \mathsf{F}$; (2) $T(v_{p_1}) = \mathsf{F}$, $T(v_{q_1}) = \mathsf{T}$, and $T(v_{r_1}) = \mathsf{F}$; or (3) $T(v_{p_1}) = \mathsf{F}$, $T(v_{q_1}) = \mathsf{F}$, and $T(v_{r_1}) = \mathsf{T}$. Suppose the first case holds (the other cases follow by symmetry). Schedule $R(v_{p_1}, \mathsf{T})$, $R(v_{q_1}, \mathsf{F})$, $R(v_{r_1}, \mathsf{F})$, $W(v_{p_1}, \mathsf{F})$, and $W(v_{q_1}, \mathsf{T})$ from $P1$; then $R(v_{q_1}, \mathsf{T})$, $R(v_{r_1}, \mathsf{F})$, $R(v_{p_1}, \mathsf{F})$, $W(v_{q_1}, \mathsf{F})$, and $W(v_{r_1}, \mathsf{T})$ from $P2$; then $R(v_{r_1}, \mathsf{T})$, $R(v_{p_1}, \mathsf{F})$, $R(v_{q_1}, \mathsf{F})$, $W(v_{r_1}, \mathsf{F})$, and $W(v_{p_1}, \mathsf{T})$ from $P3$. Note that at this point, the current "value" of each v_i is the same as it was before this step of the construction.
4. Repeat the construction of the previous two steps for clauses $C_2, \ldots, C_m$ in an analogous manner.

The reader may verify that this is a legal schedule.

Conversely, suppose $\mathcal{V}$ is a positive VSC instance. If S is a legal schedule for $\mathcal{V}$, then the second value written to each v_i will be our satisfying assignment. For $i = 1, 2, \ldots, n$, let $T(v_i) = \mathsf{T}$ if the first $W(v_i, \mathsf{F})$ from $P1$ precedes in S the first $W(v_i, \mathsf{T})$ from $P2$; otherwise, let $T(v_i) = \mathsf{F}$. A simple induction establishes that any prefix S_j of S ending in $W(z, 3j)$, $j = 1, \ldots, m$, contains precisely the following

events: all events in the prefix of $P1$ ending in $W(z, 3j-2)$, all events in the prefix of $P2$ ending in $W(z, 3j-1)$, and all events in the prefix of $P3$ ending in $W(z, 3j)$.

For $j = 1, \ldots, m$, consider the operations in $P1$, $P2$, and $P3$ in S immediately following S_j:

$$\begin{array}{lll} R(z,3j) & R(z,3j) & \\ R(v_{p_j},\mathsf{T}) & R(v_{q_j},\mathsf{T}) & R(v_{r_j},\mathsf{T}) \\ R(v_{q_j},\mathsf{F}) & R(v_{r_j},\mathsf{F}) & R(v_{p_j},\mathsf{F}) \\ R(v_{r_j},\mathsf{F}) & R(v_{p_j},\mathsf{F}) & R(v_{q_j},\mathsf{F}) \end{array} .$$

Due to these reads, it is not possible to extend S_j to a legal schedule unless the current "value" of exactly one of v_{p_j}, v_{q_j}, and v_{r_j} is T. Assume that v_{p_j} is T, while v_{q_j} and v_{r_j} are F. A careful inspection of the operations for C_j reveals that the following is a subsequence of S_{j+1} after S_j: $R(v_{p_j},\mathsf{T})$, $R(v_{q_j},\mathsf{F})$, $R(v_{r_j},\mathsf{F})$, $W(v_{p_j},\mathsf{F})$, $W(v_{q_j},\mathsf{T})$ (all from $P1$), $R(v_{q_j},\mathsf{T})$, $R(v_{r_j},\mathsf{F})$, $R(v_{p_j},\mathsf{F})$, $W(v_{q_j},\mathsf{F})$, $W(v_{r_j},\mathsf{T})$ (from $P2$), $R(v_{r_j},\mathsf{T})$, $R(v_{p_j},\mathsf{F})$, $R(v_{q_j},\mathsf{F})$, $W(v_{r_j},\mathsf{F})$, $W(v_{p_j},\mathsf{T})$ (from $P3$). This subsequence may be interleaved with the remaining operations in $S_{j+1} - S_j$, namely, $R(z,3j)$, $R(z,3j)$, $W(z,3j+1)$, $W(z,3j+2)$, $R(z,3j+1)$, $R(z,3j+2)$, and $W(z,3(j+1))$. By symmetry, these properties also hold in the case where v_{q_j} is T, while v_{r_j} and v_{p_j} are F, or the case where v_{r_j} is T, while v_{p_j} and v_{q_j} are F.

We claim that for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, the "value" of v_i through S_j is $T(v_i)$. This is established via a simple induction, where the base case, $j = 1$, holds by definition. Moreover, by the characterization of S_{j+1} of the previous paragraph, it follows by inspection that for $i = 1, \ldots, n$, the "value" of v_i through S_{j+1} is the same as through S_j.

It follows that $T(v_1), \ldots, T(v_n)$ is a satisfying one-in-three assignment for each clause in $\mathcal{F}$, and hence $\mathcal{F}$ is one-in-three satisfiable. □

The construction depicted in Figure 7 uses only $n+1$ locations. Alternatively, it can be modified to have each location assume at most two values.

2.4. Comparison with serializability. The VSC problem is reminiscent of the serializability problem for database transactions. The most similar variant is that of view serializability. In the *view serializability* problem [31], we are given a *history* H, i.e., a total order on a set of reads and writes, where each read or write is associated with a particular database transaction, and each read or write contains an address but not a value. Each read is assumed to read from the last preceding write in H to the same address. The task is to determine if there is a total order on the *transactions* that preserves this mapping of reads to writes. The view serializability problem is NP-complete [31].

The VSC problem differs from the view serializability problem in at least four ways. First, in the VSC problem, legal schedules may interleave operations from different processors: the sequence of operations at a processor must be in order but need not be consecutive in a legal schedule. For example, the instance in Figure 1 is a positive VSC instance, but a negative instance for view serializability: both $S = S_1S_2$ and $S = S_2S_1$ have reads-from violations. Second, the *input* to a view serializability problem, a consistent total order of the reads and writes, is the desired *output* of the VSC problem. Third, when a database system is correctly enforcing serializability, each transaction that is not aborted will view an unchanged database during the course of its operations. Thus a single read of an address suffices to learn the value in that location for the duration of the transaction, and in general, it may be assumed that within each transaction, there is at most one read and one write to each location.

These restrictions are not appropriate for the VSC problem since communicating processors exchange values by writing and reading memory. Fourth, the VSC problem does not provide a mapping of reads to writes that must be preserved. However, in section 4, we will consider a variant of the VSC problem (the VSC-read problem) in which the input includes such a mapping, and a legal schedule must preserve this mapping.

For the VSC problem, we observe that the ability to interleave input sequences is not all that helpful unless the number of processors is bounded. Specifically, the construction in Observation 2.3 converts any VSC instance into one in which there is a legal schedule if and only if there is a legal schedule such that each processor sequence is a consecutive subsequence of the schedule. Thus the VSC problem is NP-complete even when interleaving is not permitted. On the other hand, if the number of processors is bounded, then interleaving is quite powerful. Whereas the VSC problem with three processors is NP-complete, there is a trivial linear-time algorithm for serializability when the number of transactions is restricted to a constant k: with k "processors," there are only a constant number, $k!$, of possible serializations to check.

3. Verifying linearizability. In this section, we present results for testing linearizability. We begin with some preliminary remarks in section 3.1, including an efficient reduction from the VL problem to the VSC problem. In section 3.2, we show that the VL problem is NP-complete. (The NP-completeness proof together with the reduction can be used for an alternative proof of Corollary 2.6.) Then in section 3.3, we present a polynomial-time algorithm for the VL problem with $O(\log n)$ processors.

3.1. Preliminaries. As discussed in section 1, linearizability is more restrictive than sequential consistency, in that a legal schedule must respect the time intervals for the operations. This added constraint makes implementations of linearizability provably slower than implementations of sequential consistency [8]. In the interest of memory system performance, shared-memory multiprocessors such as the Kendall Square KSR1 [12] support sequential consistency instead of linearizability. On the other hand, linearizability has the advantage over sequential consistency that each address, or, more generally, each shared data object, can be considered in isolation. Herlihy and Wing [22] proved that a system is linearizable if and only if each object in the system is linearizable. Thus linearizable objects can be implemented, verified, and executed independently. In this paper, we use the following implication of the Herlihy and Wing theorem.

FACT 3.1. *$\mathcal{V}$ is a positive instance of the VL problem if and only if, for each address a, the subinstance of $\mathcal{V}$ comprised solely of the operations on a is a positive instance.*

Thus as indicated in Table 1, we can assume without loss of generality that a VL instance has but a single location. This assumption does not alter the complexity of the problem: if $f(n)$ is the running time on an instance of size n, then since $f(n) \geq n$, $f(n) = f(\sum_{i=1}^{k} n_i) \geq \sum_{i=1}^{k} f(n_i)$, where $n_1, \ldots, n_k$ are the respective sizes of the subinstances on each of the k locations.

Note as well that if the number of processors is not bounded, then we can also assume without loss of generality that a VL instance has but a single operation per processor, as indicated in Table 1. This follows since the intervals for operations by a single processor do not overlap, so the total order between them is enforced whether or not they are considered to be part of the same processor sequence.

A VL instance can be reduced to a VSC instance that uses additional operations that reflect the scheduling constraints defined by the start-of-interval and end-of-interval times. Specifically, we have the following reduction from the VL problem to the VSC problem.

THEOREM 3.2. *There is an $O(n \log n)$-time reduction from the VL problem with n operations and k variables to k instances of the VSC problem with two variables, at most three memory operations per sequence, and a total of $O(n)$ operations over all instances.*

Proof. By Fact 3.1, it suffices to construct distinct VSC instances for each of the k variables in the VL instance. Consider one such variable, a, and let $\mathcal{V}$ be the subinstance comprised of the operations on a, with n_a operations. Let $t_1, t_2, \ldots, t_{n_a}$ be the end-of-interval times in $\mathcal{V}$ in increasing order. We construct the following VSC instance $\mathcal{V}'$ using two types of sequences. For each operation $read(a, d, t_i, t_j)$ or $write(a, d, t_i, t_j)$ in $\mathcal{V}$, we have in $\mathcal{V}'$ the following type I sequence: (1) $read(b, \tau)$, where τ is the largest end-of-interval time in $\mathcal{V}$ less than t_i (if any); (2) $read(a, d)$ or $write(a, d)$; and (3) $write(b, t_j)$. In addition, for each end-of-interval time t_j, $j < n_a$, we have a type II sequence (1) $read(b, t_j)$, (2) $read(b, t_{j+1})$. Clearly, $\mathcal{V}'$ is a VSC instance with two variables, at most three memory operations per sequence, and $O(n_a)$ operations. We will show that the operations on b encode the scheduling constraints defined by the start-of-interval and end-of-interval times.

We begin by showing that if two operations have nonoverlapping intervals in $\mathcal{V}$, then the order between them is respected by any legal schedule for $\mathcal{V}'$.

LEMMA 3.3. *Consider any two operations π_i and π_j in $\mathcal{V}$ and the corresponding operations π_i' and π_j' in $\mathcal{V}'$. If the end-of-interval time for π_i is less than the start-of-interval time for π_j, then π_i' precedes π_j' in any legal schedule for $\mathcal{V}'$.*

Proof. Let S' be a legal schedule for $\mathcal{V}'$. Consider the subsequence S_r' of S' of all $read(b, t)$ operations in $\mathcal{V}'$. Since there is exactly one $write(b, t)$ operation for each end-of-interval t, all $read(b, t)$ operations for the given t are consecutive in S_r' and follow this $write(b, t)$ operation in S'. Moreover, due to the type II sequences, the $read(b, t)$ operations in S_r' are in nondecreasing order of t. If the end-of-interval time t_i for π_i is less than the start-of-interval time t_j for π_j, then there is a $read(b, t_k)$ operation preceding π_j' in a type I sequence such that $t_k \geq t_i$. It follows that π_i' precedes $write(b, t_i)$ precedes $read(b, t_i)$ precedes $read(b, t_k)$ precedes π_j' in S'. ∎

We now prove the correctness of our construction.

LEMMA 3.4. *$\mathcal{V}'$ is a positive VSC instance if and only if $\mathcal{V}$ is a positive VL instance.*

Proof. Suppose $\mathcal{V}$ is a positive VL instance, and let A be an assignment of times for a legal schedule for $\mathcal{V}$. For each operation π' in $\mathcal{V}'$, define $A'(\pi')$ as follows:

1. If π' is an operation on a, then $A'(\pi') = A(\pi)$, where π is the corresponding operation in $\mathcal{V}$.
2. If π' is an operation on b with data value t, then $A'(\pi') = t$.

Let S be a sequence of the operations in $\mathcal{V}'$ in nondecreasing order according to A' such that among operations with the same A' value, the operation on a (if any) precedes the write operation on b precedes any read operations on b. We claim that S is a legal schedule for $\mathcal{V}'$. First, observe that any type II sequence appears in order in S. Moreover, consider the type I sequence constructed for an operation $\pi = read(a, d, t_1, t_2)$ or $write(a, d, t_1, t_2)$ in $\mathcal{V}$: $read(b, \tau)$ (if any), $\pi' = read(a, d)$ or $write(a, d)$, and $write(b, t_2)$. By construction and since A corresponds to a legal

schedule,

$$\tau < t_1 \leq A(\pi) = A'(\pi') \leq t_2,$$

and hence the type I sequence also appears in order in S. Thus S is an interleaving of the individual sequences. Second, there are no reads-from violations on a since the order of operations on a corresponds to a legal schedule for $\mathcal{V}$. Moreover, there are no reads-from violations on b since for each $read(b, \tau)$ operation, τ is an end-of-interval time, and hence a $write(b, \tau)$ operation is the last preceding write operation in S.

Conversely, suppose $\mathcal{V}'$ is a positive VSC instance, and let S' be a legal schedule for $\mathcal{V}'$. Let $S'_a = \pi'_1, \pi'_2, \ldots, \pi'_{n_a}$ be the subsequence of S' comprised of the operations on a. Let $S_a = \pi_1, \pi_2, \ldots, \pi_{n_a}$ be the corresponding operations in $\mathcal{V}$. Since S' is legal, there are no reads-from violations in S_a. Let ϵ_0 be the minimum difference between any pair of times (start-of-interval or end-of-interval) in $\mathcal{V}$; let $\epsilon = \epsilon_0/n$. We assign times to operations in S_a inductively as follows. The first operation π_1 is assigned a time equal to its start-of-interval time. For $k = 2, \ldots, n_a$, the operation π_k in S_a is assigned a time equal to the maximum of its start-of-interval time and ϵ greater than the time assigned to π_{k-1}. We claim that even in this latter case, the time assigned to π_k is within its interval. To see this, let π_i, $1 \leq i < k$, be the last operation preceding π_k in S_a that is assigned a time equal to its respective start-of-interval time t_i. The time assigned to π_k is $t_i + (k-i)\epsilon$. Since π'_i precedes π'_k in S', it follows by Lemma 3.3 that the start-of-interval time for π_i is less than the end-of-interval time for π_k by at least ϵ_0. Thus since

$$t_i + (k-i)\epsilon < t_i + n\epsilon = t_i + \epsilon_0,$$

the time assigned to π_k is within its interval. □

As the reader may verify, this transformation can be done in $O(n \log n)$ time. Theorem 3.2 follows. □

3.2. The VL problem is NP-complete. In this section, we prove the NP-completeness of the VL problem by a reduction from the Satisfiability problem (SAT) [13]. By Fact 3.1, it is necessary to consider a reduction based on a single location. The VL problem is in NP since given an assignment of times, we can test in polynomial time that each operation is assigned a time within its interval and that the schedule defined by the assignment has no reads-from violations.

THEOREM 3.5. *The VL problem is* NP*-complete.*

Proof. Consider an instance $\mathcal{F}$ of SAT with n variables, $v_1, v_2, \ldots, v_n$, and m clauses, $C_1, C_2, \ldots, C_m$. Without loss of generality, assume that each variable and its negation appear in at least one clause, but not the same clause, and that there are no repeated variables in a clause. We construct an instance of the VL problem with at most $5nm + 4n + m$ operations, corresponding to $\mathcal{F}$. To simplify the description, the construction has multiple operations sharing the same start-of-interval or end-of-interval times; these ties can be broken arbitrarily to ensure unique time values. In particular, we use only integral times in our simplified description, with at most $m + 2$ intervals sharing the same start or end time, so unique positive rationals can be readily selected to break any ties.

Figure 8 depicts an example construction. For each clause C_j, $j = 1, \ldots, m$, we have a $read(a, c_j, 1, 3n{+}1)$ operation, denoted the *clause read* for C_j. For each variable v_i, $i = 1, \ldots, n$, assignment to v_i is simulated using two operations: $write(a, i, 3i - 1, 3i)$ and $write(a, \bar{\imath}, 3i - 1, 3i)$.

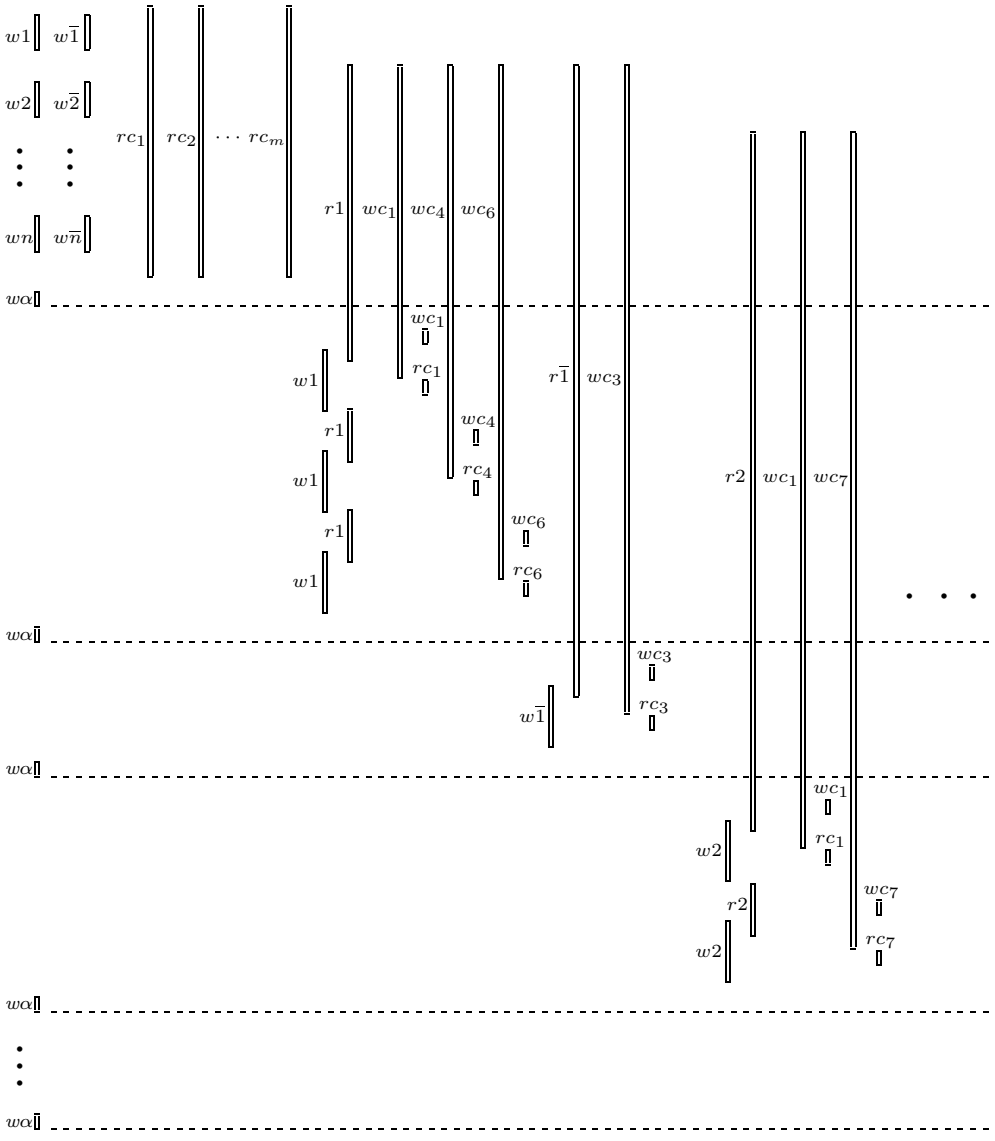

FIG. 8. *Transforming an instance of* SAT *to an instance of VL with a single location. There are* n *variables,* $v_1, \ldots, v_n$*, and* m *clauses,* $C_1, \ldots, C_m$*. The full construction has at most* $5nm+4n+m$ *operations. Here the literal* v_1 *appears in exactly clauses* C_1*,* C_4*, and* C_6*, the literal* $\overline{v}_1$ *appears in clause* C_3 *only, the literal* v_2 *appears in exactly clauses* C_1 *and* C_7*, and so forth. Every column corresponds to a processor sequence. Vertical boxes depict the intervals of time for the respective read* (r) *or write* (w) *operations to the single location; time progresses from top to bottom in the figure. The number or symbol following each* r *or* w *indicates the value read or written.*

We have $2n$ write operations, used to partition the problem into $2n$ phases, one for each literal, as follows. Consider $i = 1, \ldots, n$. Let m_i ($m_{\overline{i}}$) be the number of clauses containing the literal v_i ($\overline{v}_i$, respectively). By assumption, $m_i > 0$, $m_{\overline{i}} > 0$, and $m_i + m_{\overline{i}} \leq m$. Let $\Delta_i = (7m+4)(i-1) + 3n + 3$ and let $\Delta_{\overline{i}} = \Delta_i + 7m_i + 2$. There is a $write(a, \alpha, \Delta_i - 1, \Delta_i)$ and a $write(a, \alpha, \Delta_{\overline{i}} - 1, \Delta_{\overline{i}})$.

There is a set of operations for each literal v_i, denoted the *group* of operations for i. For $i = 1, \ldots, n$, if $C_{i_1}, C_{i_2}, \ldots, C_{i_{m_i}}$ are the clauses containing the literal v_i, we have the following $5m_i$ intervals. First, for C_{i_1}, we have

- $ri^{(1)} = read(a, i, 3i + 1, \Delta_i + 4)$,
- $wi^{(1)} = write(a, i, \Delta_i + 3, \Delta_i + 7)$,
- $wc_{i_1}^{(1)} = write(a, c_{i_1}, 3i + 1, \Delta_i + 5)$,
- $wc_{i_1}^{(2)} = write(a, c_{i_1}, \Delta_i + 1, \Delta_i + 2)$,
- $rc_{i_1} = read(a, c_{i_1}, \Delta_i + 5, \Delta_i + 6)$.

Then for C_{i_k}, $k = 2, \ldots, m_i$, we have

- $ri^{(k)} = read(a, i, \Delta_i + 7(k-1), \Delta_i + 7(k-1) + 4)$,
- $wi^{(k)} = write(a, i, \Delta_i + 7(k-1) + 3, \Delta_i + 7(k-1) + 7)$,
- $wc_{i_k}^{(1)} = write(a, c_{i_k}, 3i + 1, \Delta_i + 7(k-1) + 5)$,
- $wc_{i_k}^{(2)} = write(a, c_{i_k}, \Delta_i + 7(k-1) + 1, \Delta_i + 7(k-1) + 2)$,
- $rc_{i_k} = read(a, c_{i_k}, \Delta_i + 7(k-1) + 5, \Delta_i + 7(k-1) + 6)$.

The operations $wc_{i_1}^{(1)}, wc_{i_2}^{(1)}, \ldots, wc_{i_{m_i}}^{(1)}$ are denoted the *clause writes* for v_i.

Likewise, there is a set of operations for each literal $\overline{v}_i$, denoted the *group* of operations for $\bar{\imath}$. For $i = 1, \ldots, n$, if $C_{\bar{i}_1}, C_{\bar{i}_2}, \ldots, C_{\bar{i}_{m_{\bar{i}}}}$ are the clauses containing the literal $\overline{v}_i$, we have the $5m_{\bar{i}}$ intervals obtained from the previous definition by replacing $\Delta_{\bar{\imath}}$ with Δ_{i+1} and leaving "$3i + 1$" unchanged but otherwise replacing i with $\bar{\imath}$ throughout.

This completes the construction.

The idea behind the construction is the following. Consider a variable v_1, and refer to Figure 8. Recall that all reads and writes are to the same location. Both $w1$ and $w\overline{1}$ on the left must be scheduled before any $r1$ or $r\overline{1}$; the second write scheduled corresponds to the truth assignment. If $w1$ is scheduled second, then the first $r1$ can be scheduled, followed by the clause writes wc_1, wc_4, and wc_6 for v_1. If all clauses can be satisfied by the truth setting, then the set of all clause writes will ensure that all m clause reads can be scheduled during their common interval. Consider the three $w1$ operations and the two $r1$ operations below the first dashed line, together with the $r1$ operation that crosses this line. Since the crossing $r1$ has been scheduled, the two $r1$ operations below the line can pair up with the first two $w1$ operations; this in turn permits the rc_1 (below the line) to be paired with the wc_1 directly above it. On the other hand, if $w\overline{1}$ had been scheduled second instead, then the first $w1$ is paired with the crossing $r1$. Hence the first $w1$ is scheduled between the intervals for the wc_1 and the rc_1 below the line; this implies that a clause write wc_1 is needed below the line. For this reason, all clause writes for a literal not in the truth assignment must be scheduled below the first dashed line. It follows that all clause reads can be scheduled if and only if we have a satisfying truth assignment.

LEMMA 3.6. *Let $\mathcal{F}$ be an instance of a* SAT *problem, and let $\mathcal{V}$ be the instance of the VL problem constructed as described above. Then $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable. We will construct a schedule in which, for each v_i, the latter of $write(a, i, 3i - 1, 3i)$ and $write(a, \bar{\imath}, 3i - 1, 3i)$ scheduled corresponds to the satisfying truth assignment. Let $T(v_1), \ldots, T(v_n)$, where $T(v_i) \in \{\mathsf{T}, \mathsf{F}\}$, be a satisfying assignment for $\mathcal{F}$. We construct the following schedule for $\mathcal{V}$:

1. Repeat the following for $i = 1, 2, \ldots, n$:
   If $T(v_i) = \mathsf{T}$, schedule $write(a, \bar{\imath}, 3i - 1, 3i)$ at time $3i - 1$, then $write(a, i, 3i - 1, 3i)$ at time $3i$. Consider the $5m_i$ operations in the group for i, together with

the m_i corresponding clause reads, $read(a, c_{i_1}, 1, 3n+1)$ through $read(a, c_{i_{m_i}}, 1, 3n+1)$. Schedule $ri^{(1)} = read(a, i, 3i+1, \Delta_i+4)$ at time $3i+1$. Then after time $3i+1$ and before time $3i+2$, for $k = 1, \ldots, m_i$, schedule each of m_i clause writes paired with any unscheduled, corresponding clause reads: $wc_{i_k}^{(1)} = write(a, c_{i_k}, 3i+1, \Delta_i + 7(k-1) + 5)$ followed by, if it has not yet been scheduled, $read(a, c_{i_k}, 1, 3n+1)$.

The case where $T(v_i) = \mathsf{F}$ is symmetric, and left to the reader.

2. Since $T(v_1), \ldots, T(v_n)$ is a satisfying assignment for $\mathcal{F}$, all clause reads have been scheduled by this point. Schedule $write(a, \alpha, \Delta_1 - 1, \Delta_1)$ at time Δ_1.
3. Repeat the following for $i = 1, 2, \ldots, n$:
   If $T(v_i) = \mathsf{T}$, consider the $4m_i - 1$ unscheduled operations in group i and the $5\overline{m}_i$ unscheduled operations in group $\bar{\imath}$:
   (a) Schedule $wc_{i_1}^{(2)}$ at time $\Delta_i + 2$, then rc_{i_1} at time $\Delta_i + 5$, then $wi^{(1)}$ at time $\Delta_i + 7$.
   (b) Then repeat for $k = 2, \ldots, m_i$: Schedule $ri^{(k)}$ at time $\Delta_i + 7(k-1) + 1$, then $wc_{i_k}^{(2)}$ at time $\Delta_i + 7(k-1) + 2$, then rc_{i_k} at time $\Delta_i + 7(k-1) + 5$, and finally $wi^{(k)}$ at time $\Delta_i + 7(k-1) + 7$.
   (c) This completes group i. Schedule $write(a, \alpha, \Delta_{\bar{\imath}} - 1, \Delta_{\bar{\imath}})$ at time $\Delta_{\bar{\imath}}$.
   (d) Schedule $wc_{\bar{\imath}_1}^{(2)}$ at time $\Delta_{\bar{\imath}} + 1$, then $w\bar{\imath}^{(1)}$ at time $\Delta_{\bar{\imath}} + 3$, then $r\bar{\imath}^{(1)}$ at time $\Delta_{\bar{\imath}} + 4$, then $wc_{\bar{\imath}_1}^{(1)}$ at time $\Delta_{\bar{\imath}} + 5$, and finally $rc_{\bar{\imath}_1}$ at time $\Delta_{\bar{\imath}} + 6$.
   (e) Then repeat for $k = 2, \ldots, m_{\bar{\imath}}$: Schedule $wc_{\bar{\imath}_k}^{(2)}$ at time $\Delta_{\bar{\imath}} + 7(k-1) + 1$, then $w\bar{\imath}^{(k)}$ at time $\Delta_{\bar{\imath}} + 7(k-1) + 3$, then $r\bar{\imath}^{(k)}$ at time $\Delta_{\bar{\imath}} + 7(k-1) + 4$, then $wc_{\bar{\imath}_k}^{(1)}$ at time $\Delta_{\bar{\imath}} + 7(k-1) + 5$, and finally $rc_{\bar{\imath}_k}$ at time $\Delta_{\bar{\imath}} + 7(k-1) + 6$.
   (f) This completes group $\bar{\imath}$. Schedule $write(a, \alpha, \Delta_{i+1} - 1, \Delta_{i+1})$ at time Δ_{i+1}.

   The case where $T(v_i) = \mathsf{F}$ is symmetric and left to the reader.

The reader may verify that this is a legal schedule.

Conversely, suppose $\mathcal{V}$ is a positive instance. If $\mathcal{S}$ is a legal schedule for $\mathcal{V}$, then let $\mathcal{T}$ be the truth assignment defined as follows: for each v_i, if $write(a, \bar{\imath}, 3i-1, 3i)$ precedes $write(a, i, 3i-1, 3i)$ in $\mathcal{S}$, then $\mathcal{T}(v_i) = \mathsf{T}$; otherwise, $\mathcal{T}(v_i) = \mathsf{F}$.

We observe the following for $i = 1, \ldots, n$: Both $write(a, i, 3i-1, 3i)$ and $write(a, \bar{\imath}, 3i-1, 3i)$ precede both $ri^{(1)}$ and $r\bar{\imath}^{(1)}$ in $\mathcal{S}$. Thus if $\mathcal{T}(v_i) = \mathsf{F}$, then while $r\bar{\imath}^{(1)}$ may be scheduled in $\mathcal{S}$ prior to time $3i+3$, $ri^{(1)}$ cannot be. The $ri^{(1)}$ operation must be scheduled after the only other operation that writes i prior to the end of its interval, namely $wi^{(1)}$. Hence $wc_{i_1}^{(2)}$ precedes $wi^{(1)}$ precedes $ri^{(1)}$ precedes rc_{i_1}. But this implies that for $k = 2, \ldots, m_i$, $wc_{i_k}^{(2)}$ precedes $wi^{(k)}$ precedes $ri^{(k)}$ precedes rc_{i_k}. The case where $\mathcal{T}(v_i) = \mathsf{T}$ is symmetric.

Suppose $\mathcal{T}$ does not satisfy a clause C_j. Consider the literals in C_j sorted by their index, and let x be the number of literals in C_j. For $k = 1, \ldots, x$, define $j_k \in \{1, \bar{1}, 2, \bar{2}, \ldots, n, \bar{n}\}$ such that $j_k = i$ (or $\bar{\imath}$) if and only if v_i ($\overline{v}_i$, respectively) is the kth literal in C_j. We claim that all writes of c_j are scheduled in $\mathcal{S}$ after Δ_1. The proof is by induction on decreasing k. Consider the last rc_j in $\mathcal{S}$, in group j_x. Due to the argument given in the preceding paragraph, there is only one write of c_j that could have been scheduled so as to be read by the rc_j: the sole write of c_j in group j_x whose interval is not strictly after Δ_{j_x}. Assume inductively that all writes of c_j from groups j_{k+1} to j_x are scheduled after $\Delta_{j_{k+1}}$. Thus due to the argument given in the preceding paragraph, there is only one write of c_j that could have been scheduled so

as to be read by the rc_j: the sole write of c_j in group j_k whose interval is not strictly after Δ_{j_k}. The claim follows by induction.

Since there are no writes of c_j in $\mathcal{S}$ until after $\Delta_1 = 3n + 3$, but the clause read $read(a, c_j, 1, 3n + 1)$ must be scheduled before Δ_1, $\mathcal{S}$ is not a legal schedule, a contradiction.

Hence $\mathcal{T}$ is a satisfying assignment for $\mathcal{F}$. □

Since the above transformation can be done in polynomial time, Theorem 3.5 follows. □

3.3. The VL problem with few processors. In this section, we describe a polynomial-time algorithm for the VL problem when the number of processors (i.e., sequences) is bounded. Recall that if two operations have nonoverlapping intervals, their order in any legal schedule is predetermined. Thus since we can consider each address in isolation, the number of possible schedules grows with the number of overlapping intervals for operations on that address. The specific result we obtain is the following.

THEOREM 3.7. *The VL problem restricted to instances such that at any time t, there are at most k operations on the same address whose intervals contain t, can be solved in $O(n2^{O(k)} + n \log n)$ time.*

Proof. We consider each address x in isolation. Let S_x be the set of operations on x. Let $t_0 > 0$ be the earliest start-of-interval time and t_∞ be the latest end-of-interval time for an operation in S_x. To simplify the discussion that follows, we augment S_x with two additional writes to x: Let $S'_x = S_x \cup \{write(x, \delta, t_0/3, t_0/2), write(x, \delta, t_\infty + 1, t_\infty + 2)\}$, where δ is a value not appearing in S_x. For each address, we may assume without loss of generality that there are only k processors.

Let G_x be a leveled acyclic digraph, with one level for each operation in S'_x, in order of increasing end-of-interval times, and at most $k2^{k-1}$ vertices per level, as follows. We identify each level by the finishing time t of its corresponding operation α_t. There are at most $k - 1$ additional operations in S'_x with intervals containing t; these may or may not be assigned times greater than t. Consider all possible subsets of these operations; there are at most 2^{k-1} of them. Each vertex at level i specifies one of these subsets, plus a last write to x. The vertex represents a prefix of a schedule of the operations in S'_x, namely, a schedule in which the operations in the subset as well as all operations with start-of-interval times after t are assigned times later than t, and the last write assigned a time no later than t is indicated. Since for a given subset there are at most k possible choices of a last write, we have at most $k2^{k-1}$ nodes. The first level consists of a single node, with last write $write(x, \delta, t_0/3, t_0/2)$; the last level consists of a single node, with last write $write(x, \delta, t_\infty + 1, t_\infty + 2)$.

The edges of G_x are defined as follows. Consider two consecutive levels t and t', and let α and α' be the operations with end-of-interval times t and t', respectively. All intervals, other than α's, that contain time t must contain t'. Consider two vertices in G_x, v at level t and v' at level t'. There is an edge between v and v' if the following conditions hold:

1. Every interval containing both t and t' that was assigned a time no later than t, according to v, is also assigned a time no later than t', according to v'.
2. Let A be the set of operations with intervals containing t' (including possibly α') that were assigned a time later than t (either they do not contain t or they contain t but were assigned a time later than t, according to v) but were assigned a time no later than t', according to v'. Every read in A must either read the value of the last write in v or has a corresponding write of the same value in A. Moreover, the last write in v' must either be in the set A or, if

there are no writes in A, the same last write as in v.

Note that the operations in A can be safely scheduled without reads-from violations in the interval between the last start-of-interval time in A and t'. It follows that there is a directed edge from v to v' if and only if there is a schedule consistent with both v and v'. This leads to the following claim, whose proof is left to the reader.

CLAIM 3.8. *We have a positive VL instance if and only if there is a source-to-sink path in G_x for each address x.*

To construct the graphs for each address, first we sort the operations by address, and within an address, by their starting and finishing times (each operation appears twice in this sorted sequence). Then we test for each pair of vertices on consecutive levels whether a directed edge should be between them. Next, we test for a source-to-sink path in each G_x using depth-first search. If we have a positive instance, we assign times to the operations based on the information in the vertices visited along the source-to-sink paths. Each G_x can be constructed and searched in $O(n_x 2^{O(k)})$ time, where n_x is the number of operations in S_x. Thus the total running time is $O(n \log n + n2^{O(k)})$. □

Since with k processors there can be at most k overlapping intervals, Theorem 3.7 implies, for instance, the following two corollaries.

COROLLARY 3.9. *There is an $O(n \log n)$-time algorithm for the VL problem with any fixed number of processors.*

COROLLARY 3.10. *There is a polynomial-time algorithm for the VL problem with $O(\log n)$ processors.*

4. Providing additional input information. In the previous sections, we have considered the basic VSC and VL problems, in which the only constraints on legal schedules arise from (1) either the order of operations at a processor or the time intervals, and (2) the fact that some write operation with the same address and value must precede each read operation, with no intervening write with the same address but a different value. In this section, we consider variants of the VSC and VL problems in which the input provides additional information on either the pairing of reads to particular writes (section 4.1), the order of writes to a location (section 4.2), the old value overwritten by each write (section 4.3), or the order of all conflicting operations to a location (section 4.4). As shown in Table 1, this additional information helps in some cases, but not in others. We also show how our algorithmic results can be extended to handle atomic read-modify-write operations, with no asymptotic penalty (section 4.5).

Each of the variants considered in this section is motivated by practical considerations in existing multiprocessors. For further details, we refer the reader to [17, 19].

We require that the additional information provided by the memory system as input in these variants be respected in any legal schedule. What happens if the additional information provided is incorrect? If there are anomalies in the additional information, e.g., a read is paired with a write with a different address, we detect this and report a negative instance. If the additional information is incorrect such that a positive instance (ignoring the information) becomes a negative instance, we report a negative instance: there is clearly something wrong with the memory system. The most important property that we require, however, is that the testing procedure must never be persuaded by incorrect (or even correct) additional information that an execution was sequentially consistent or linearizable when in fact it was not. Conversely, if both the execution and the additional information were correct, the testing procedure must report a positive instance.

4.1. Providing the read-mapping. In the basic VSC and VL problems, whenever there are multiple writes with the same address and value, there may be ambiguity as to which of the writes is to be paired with a given read of the same address and value. The question addressed in this section is as follows: If for each read there is no ambiguity as to with which write it is to be paired, do the VSC and VL problems remain NP-complete?

We define the *VSC-read* and *VL-read* problems, in which for each read operation, it is known precisely which write was responsible for the value read; a legal schedule must respect this relation. (A schedule S respects the relation if and only if, for each read in S, the write to which it is mapped is the last preceding write in S with the same address.) The function mapping each read to the responsible write is called a *read-mapping*. A schedule that does not respect the read-mapping has a *reads-from violation*.

VSC-read. We will show that the VSC-read problem is NP-complete by a reduction from view serializability. Recall that in the *view serializability* problem, we are given a *history* H, i.e., a total order on a set of reads and writes, where each read or write is associated with a particular database transaction, and each read or write contains an address but not a value. Each read is assumed to read from the last preceding write in H to the same address. The task is to determine if there is a total order on the *transactions* that preserves the read-mapping in H.

THEOREM 4.1. *The VSC-read problem restricted to instances in which each sequence contains at most three memory operations is* NP*-complete.*

Proof. We begin by showing a reduction to the VSC-read problem with no restrictions on the number of operations in a sequence. Given a history H, an instance of a view serializability problem, we construct an instance of the VSC-read problem as follows. Let α 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 last previous write in H to the same address. Let $S_i = W(\alpha, i)S_i'R(\alpha, i)$ for all transactions i. This construction ensures that all operations in S_i must be scheduled consecutively in any legal schedule: any schedule that interleaves operations from different S_i's must violate the reads-from mapping for α. It follows that we have a positive VSC-read instance if and only if H is view serializable.

To complete the theorem, we note that Observation 2.3 can be adapted to the VSC-read problem by simply adding the read-mapping. □

VL-read. We now turn to the VL-read problem and show that, in contrast to the VSC-read problem, there is an $O(n \log n)$-time algorithm for this problem.

THEOREM 4.2. *There is an $O(n \log n)$-time algorithm for the VL-read problem.*

Proof. We sort the input by address, and within an address, by start-of-interval and end-of-interval times. We check to see that each read is mapped to a write operation with the same address and value; otherwise, we have a negative instance. Consider each address separately. Define a *cluster* to be a write w and the set R of reads mapped to the write. The write w must be assigned a time earlier than that of any read in R. Thus the start-of-interval time for w must be earlier than the end-of-interval time for any read in R; otherwise, we have a negative instance. Any legal time assignment defines an interval for a cluster from the time assigned to w to the time assigned to the last read in R; only operations in this cluster can be scheduled during this time interval. Define a *zone* for a cluster to be the interval from the earliest end-of-interval time for an operation in the cluster to the latest start-of-interval time

for an operation in the cluster. In the normal scenario, the former is earlier than the latter, and we have a *forward* zone; otherwise, we have a *backward* zone. For any legal time assignment, the interval of time for a cluster with a forward zone must contain that zone. Therefore, if two forward zones overlap, we have a negative instance.

For a cluster with a backward zone, the backward zone is the intersection of the individual operation intervals for operations in the cluster. Thus all operations in a cluster can be safely scheduled in *any* subinterval of the zone of positive length. Moreover, one can see that the interval of time for the cluster in any legal time assignment must intersect its backward zone. It follows that if a backward zone is contained within the forward zone for some other cluster, we have a negative instance.

Finally, if none of the illegal configurations described above occurs, we have a positive instance. Let ϵ_0 be the minimum difference between any pair of times in the instance. Augment each forward zone by $\epsilon_0/3$ before and after the zone. Then the operations in forward zone clusters are safely scheduled within their respective augmented zones, and the operations in backward zone clusters are safely scheduled outside all augmented forward zones using nonoverlapping subintervals: for each cluster, w is scheduled at the start of the augmented zone or subinterval, and then the operations in R are scheduled in order of start-of-interval times. The reader may verify that this is a legal schedule.

The final legal schedule is obtained by merging the individual schedules for each address. After the initial sorting, the algorithm runs in linear time. □

4.2. Providing the write-order. A second source of ambiguity in the basic VSC and VL problems is that there may be multiple writes to the same location by different processors. The question addressed in this section is as follows: If for each location there is no ambiguity as to the order of writes to the location, do the VSC and VL problems remain NP-complete?

We define the *VSC-write* and *VL-write* problems, in which for each shared-memory location, a total order on the write operations to the location is known; a legal schedule must respect this *write-order* relation.

VSC-write. We show that the VSC-write problem is NP-complete. In fact, we prove that the VSC problem is NP-complete even when each location is written to by only a single processor, yielding the following stronger result.

THEOREM 4.3. *The VSC and VSC-write problems restricted to instances in which each location is written to by only a single processor are* NP*-complete.*

Proof. Our reduction from 3SAT is depicted in Figure 9. The construction consists of operations used to select a truth assignment and then to test each clause. We use the $W(ok)$ operation in the first sequence to signal when all clauses have been tested and it is time to clean up so that the set of operations not used in testing the particular assignment may be safely scheduled if and only if the assignment satisfied the 3SAT instance. Note that for each location, all writes to that location appear in the same sequence. Thus for the VSC-write problem, the write-order is implied by the order of the individual sequences.

In all three NP-completeness proofs of section 2, the constructions have, for each variable, two writes to the same location such that the order between the two writes determines the truth setting for the variable. Here the order on two such writes is predetermined by the order in which they appear in their processor sequence. Hence we use two writes to *different* locations, which must somehow be coupled so that only one setting of the 3SAT variable occurs prior to the cleanup. The first interesting part of the construction, then, is the four sequences for each variable.

First, we have the following three sequences:

$W(a_1, 1)$	$W(b_1, 1)$	$W(c_1, 1)$
$W(a_1, 2)$	$W(b_1, 2)$	$W(c_1, 2)$
$W(a_2, 1)$	$W(b_2, 1)$	$W(c_2, 1)$
$W(a_2, 2)$	$W(b_2, 2)$	$W(c_2, 2)$
:	:	:
$W(a_m, 1)$	$W(b_m, 1)$	$W(c_m, 1)$
$W(a_m, 2)$	$W(b_m, 2)$	$W(c_m, 2)$
$R(f_1)$	$W(f_1)$	$W(f_2)$
$R(f_2)$		
$W(ok)$	$R(ok)$	$R(ok)$
$W(a_1, 1)$	$W(b_1, 1)$	$W(c_1, 1)$
$W(a_2, 1)$	$W(b_2, 1)$	$W(c_2, 1)$
:	:	:
$W(a_m, 1)$	$W(b_m, 1)$	$W(c_m, 1)$

Then for each variable v_i, $i = 1, 2, \ldots, n$, we have the following four sequences:

$W(y_i, 1)$	$W(z_i, 1)$	$R(y_i, 2)$	$R(z_i, 2)$
$W(y_i, 2)$	$W(z_i, 2)$	$R(z_i, 1)$	$R(y_i, 1)$
$W(y_i, 1)$	$W(z_i, 1)$	$W(v_i)$	$W(\overline{v}_i)$
		$R(ok)$	$R(ok)$
		$R(y_i, 2)$	$R(z_i, 2)$

Finally, for each clause $C_j = \lambda_{j,1} \vee \lambda_{j,2} \vee \lambda_{j,3}$, $j = 1, 2, \ldots, m$, where $\lambda_{j,1}$, $\lambda_{j,2}$, and $\lambda_{j,3}$ denote literals from $\{v_1, \ldots, v_n, \overline{v}_1, \ldots, \overline{v}_n\}$, we have the following three sequences:

$R(\lambda_{j,1})$	$R(\lambda_{j,2})$	$R(\lambda_{j,3})$
$R(b_j, 1)$	$R(c_j, 1)$	$R(a_j, 1)$
$R(a_j, 2)$	$R(b_j, 2)$	$R(c_j, 2)$

FIG. 9. *Transforming an instance of* 3SAT *with n variables and m clauses to an instance of VSC with $4n + 3m + 3$ sequences and locations, such that each location is written to by only a single processor. For locations that are written to only once in the entire construction, we omit the data field in the write and the reads for that location.*

We begin by proving the following lemma about these four sequences.

LEMMA 4.4. *Consider the four sequences for a variable v_i, together with an additional sequence comprised solely of a $W(ok)$ operation. Then we have the following:*

1. *There exists a legal schedule for these sequences such that $W(v_i)$ precedes $W(ok)$ precedes $W(\overline{v}_i)$.*
2. *There exists a legal schedule for these sequences such that $W(\overline{v}_i)$ precedes $W(ok)$ precedes $W(v_i)$.*
3. *There exists no legal schedule for these sequences such that both $W(v_i)$ and $W(\overline{v}_i)$ precede $W(ok)$.*

Proof. For claim 1, the reader may verify that the following is a legal schedule: $W(y_i, 1)$, $W(y_i, 2)$, $R(y_i, 2)$, $W(z_i, 1)$, $R(z_i, 1)$, $W(v_i)$, $W(ok)$, $R(ok)$, $R(y_i, 2)$, $W(z_i, 2)$, $R(z_i, 2)$, $W(y_i, 1)$, $R(y_i, 1)$, $W(\overline{v}_i)$, $R(ok)$, $R(z_i, 2)$, $W(z_i, 1)$. Claim 2 follows by symmetry. As for claim 3, suppose both did precede $W(ok)$ in a legal schedule S.

Let $S = S_1S_2$, where S_1 is the prefix of S up to and including the $W(ok)$ operation. Then the first $R(y_i, 2)$ and the first $R(z_i, 2)$ are in S_1. By symmetry, assume without loss of generality that the former precedes the latter in S_1. Then it can be seen that the first $W(y_i, 1)$ precedes $W(y_i, 2)$ which precedes $R(y_i, 1)$ in S_1. So to avoid a reads-from violation, $W(y_i, 2)$ must precede the second $W(y_i, 1)$ which must precede $R(y_i, 1)$ in S_1. But since the second $R(y_i, 2)$ is in S_2 and there are no other $W(y_i, 2)$ operations to schedule after the $R(y_i, 1)$ operation, S is not a legal schedule, and we have a contradiction. □

Likewise, in the previous construction, we simulate an OR using three writes with the same address and value, any one of which signals that a clause has been satisfied by the truth assignment. Here the order on any such writes is predetermined, and hence this approach for simulating an OR does not work. The second interesting part of the construction, then, is how the first three sequences together with all the clause sequences accurately test each clause.

Consider the jth clause C_j and its three sequences, say S_1, S_2, and S_3. There is a row in the first three sequences in which writes to a_j, b_j, and c_j set the values of these locations to 1, followed by a row in which writes to these same locations set the values to 2. These six operations must be scheduled prior to the cleanup. If C_j is satisfied, then for at least one of S_1, S_2, or S_3, the entire sequence can be scheduled prior to the cleanup. On the other hand, if C_j is not satisfied, then all operations in S_1, S_2, and S_3 remain to be scheduled during the cleanup. The second reads in S_1, S_2, and S_3 read the value 1; thus before the first such read, e.g., $R(c_j, 1)$, can be scheduled, we must schedule the appropriate write of 1, $W(c_j, 1)$, during the cleanup. But then for that particular location c_j, the read of 2 remains to be scheduled, and yet the value cannot be reset to 2.

LEMMA 4.5. *Let $\mathcal{F}$ be an instance of the* 3SAT *problem, and let $\mathcal{V}$ be the instance of the VSC-write problem constructed as depicted in Figure* 9. *Then $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable, and let $\mathcal{T}$ be a satisfying assignment for $\mathcal{F}$. We construct the following schedule for $\mathcal{V}$:

1. For each variable v_i set to true (respectively, false) by $\mathcal{T}$, schedule operations in the four sequences for v_i according to claim 1 (respectively, claim 2) of Lemma 4.4, up to but not including the $W(ok)$ added by the lemma.
2. For each clause C_j in turn, schedule as follows: First, schedule one of the reads that agrees with $\mathcal{T}$. This leaves two reads in the same sequence, say $R(c_j, 1)$ and $R(b_j, 2)$, if the second of the sequences for C_j agrees with $\mathcal{T}$. Schedule the first $W(a_j, 1)$, the first $W(b_j, 1)$, and the first $W(c_j, 1)$ (in a row in the figure), then the read $R(c_j, 1)$. Then schedule $W(a_j, 2)$, $W(b_j, 2)$, $W(c_j, 2)$, and $R(b_j, 2)$.
3. Schedule $W(f_1)$, $R(f_1)$, $W(f_2)$, $R(f_2)$, and $W(ok)$. Then schedule the two $R(ok)$ operations (from the same sequences as the $W(f_1)$ and $W(f_2)$).
4. For each variable v_i set to true (respectively, false) by $\mathcal{T}$, schedule all remaining operations in the four sequences for v_i according to claim 1 (respectively, claim 2) of Lemma 4.4 after the $W(ok)$ added by the lemma.
5. At this point, both $W(v_i)$ and $W(\overline{v}_i)$ have been scheduled, so all remaining $R(v_i)$ and $R(\overline{v}_i)$ operations can safely be scheduled next.
6. For each clause C_j in turn, schedule as follows: There are two pairs of unscheduled reads in the sequences for C_j, say $R(b_j, 1)$ followed by $R(a_j, 2)$ and $R(a_j, 1)$ followed by $R(c_j, 2)$ (if the pair $R(c_j, 1)$ and $R(b_j, 2)$ were the ones

scheduled above). Note that the last writes to a_j, b_j, and c_j scheduled each wrote the value 2. Schedule the second $W(b_j, 1)$, then $R(b_j, 1)$, then $R(a_j, 2)$. Then schedule the second $W(a_j, 1)$, then $R(a_j, 1)$, then $R(c_j, 2)$, and finally the second $W(c_j, 1)$.

The reader may verify that this is a legal schedule.

Conversely, suppose $\mathcal{V}$ is a positive instance, and let S be a legal schedule for $\mathcal{V}$. Let $S = S_1 S_2$, where S_1 is the prefix of S up to and including $W(ok)$ and S_2 is the remaining suffix of S. We observe that the claims of Lemma 4.4 apply to S since, outside of $W(v_i)$, $W(\overline{v}_i)$, and $R(ok)$, the four sequences for each v_i contain addresses appearing only in these four sequences. Let $\mathcal{T}$ be the partial truth assignment such that for each v_i, v_i is set to true if $W(v_i)$ is in S_1, v_i is set to false if $W(\overline{v}_i)$ is in S_1, and v_i is not set otherwise. (It follows from claim 3 of Lemma 4.4 that no variable is set to both true and false.) We claim that the partial truth assignment $\mathcal{T}$ satisfies $\mathcal{F}$. Suppose not, and let C_j be an unsatisfied clause. Since for each of the three sequences for C_j, the first read cannot be in S_1, then the last two reads in each such sequence are in S_2. Moreover, the last write to a_j, b_j, and c_j in S_1 wrote the value 2. By symmetry, assume without loss of generality that $R(a_j, 1)$ precedes both $R(b_j, 1)$ and $R(c_j, 1)$ in S_2. Then the second $W(a_j, 1)$ precedes $R(a_j, 1)$, to avoid a reads-from violation, which precedes $R(b_j, 1)$ which precedes $R(a_j, 2)$ in S_2. Since there is no $W(a_j, 2)$ in between the $W(a_j, 1)$ and the $R(a_j, 2)$ in S_2, S is not a legal schedule, a contradiction. Therefore, all clauses are satisfied by T. □

Since the above transformation can be done in polynomial time, Theorem 4.3 follows. □

VL-write. We now turn to the VL-write problem, and show that, in contrast to the VSC-write problem, there is an $O(n \log n)$-time algorithm for this problem.

THEOREM 4.6. *There is an $O(n \log n)$-time algorithm for the VL-write problem.*

Proof. Let $\mathcal{V}$ be an instance of the VL-write problem. We sort the instance by address and, within each address, by start-of-interval and end-of-interval times. We check to see that each write is included only in the write-order for its address; otherwise, we have a negative instance.

Consider the set of operations S on an address a in $\mathcal{V}$. Let $w_1, w_2, \ldots, w_m$ be the sequence of writes to address a as ordered by the write-order. The algorithm proceeds in rounds. We begin with all operations in S unscheduled and maintain the invariant that all the start-of-interval times for scheduled operations are less than all the end-of-interval times of unscheduled operations. At round i, if the start-of-interval time for w_i is greater than the end-of-interval time for an unscheduled operation, then we have a negative instance and halt. Otherwise, schedule w_i. Then greedily schedule, in order of increasing start-of-interval times, all unscheduled reads of the same value whose start-of-interval times are less than the end-of-interval time for all unscheduled operations. Continue on to the next round.

If any reads remain unscheduled after round m, then we have a negative instance and halt. Otherwise, a legal assignment of times for the operations in S is obtained inductively as follows. Let ϵ_0 be the minimum difference between any pair of times (start-of-interval or end-of-interval) in $\mathcal{V}$; let $\epsilon = \epsilon_0/n$. The first operation scheduled, w_1, is assigned a time equal to its start-of-interval time. Each subsequent operation is assigned a time equal to the maximum of its start-of-interval time and ϵ greater than the time assigned to the previous scheduled operation.

We now show that this greedy algorithm finds a legal schedule of S if and only if one exists. The schedule produced by the algorithm is legal since it contains all of

the operations in S, the write-order is respected, the last write scheduled prior to a read has the same value, and all operations are assigned times within their intervals. To see that this last condition holds, consider an operation π that is assigned a time greater than its start-of-interval time. Let π' be the last operation prior to π that is assigned a time equal to its respective start-of-interval time t'. Since π' is scheduled before π, t' is less than the end-of-interval time for π. Thus π is assigned a time less than $t' + \epsilon_0$ and hence within its interval.

Conversely, assume that there is a legal schedule of S. Suppose that the greedy algorithm completes $j \leq m$ rounds. For $i = 1, \ldots, j$, we claim that the set of operations S_i in the greedy schedule prior to w_i is a superset of the set of operations scheduled prior to w_i in any legal schedule of S. The proof is by induction, with a trivial basis since w_1 is the first operation in any legal schedule. Assume that the claim is true for $i-1$. Consider a legal schedule of S, and let S'_i be the set of operations prior to w_i in the schedule. Suppose there is an operation α in S'_i that is not in S_i. Since the order on writes is fixed, S'_i and S_i contain the same writes, as do S'_{i-1} and S_{i-1}. It follows from the inductive assumption that α is a read of the value written by w_{i-1}. Since α is not in S_i, there must be an operation not in S_i whose end-of-interval time is less than the start-of-interval time for α such that the operation is a read of a different value. Since α is in S'_i but reads a different value, it must be in S'_{i-1}. But then by the inductive assumption, it is in S_{i-1}, a contradiction.

From this claim, we see that any unscheduled operation in S_j will be scheduled after w_j in any legal schedule, and hence its end-of-interval time is greater than the start-of-interval time for w_j. Moreover, the set of reads not in S_m is a subset of the set of reads scheduled after w_m in any legal schedule. It follows that the algorithm successfully completes all m rounds and finds a legal schedule of S.

If for all addresses, the algorithm succeeds in finding a legal schedule, we have a positive instance. The final legal schedule for $\mathcal{V}$ is obtained by merging the individual schedules for each address. □

4.3. Read&write only. In the VSC-write and VL-write problems, the input provides a mapping from each write to the previous write on the same address (if any). In this section, we consider the complexity of the VSC and VL problems when for each write, the input provides not the identity of the previous write but only its value.

We view the memory operations as atomic read–modify–write operations. Accordingly, we define a *read&write*$(a, d_{old} : d_{new}, t_1, t_2)$ operation, where $a \in A$ is the address, $d_{old} \in D$ is the old value (returned by the read), and $d_{new} \in D$ is the new value written. A legal schedule must respect this pairing of old and new values: each *read&write*$(a, d_{old} : d_{new}, t_1, t_2)$ operation must be preceded by an operation that writes d_{old} to a with no intervening operation that writes a different value to a. As before, the start-of-interval time t_1 and end-of-interval time t_2 are needed for the VL problem but not the VSC problem. In the VSC and VL problems with *read&write only*, all operations are *read&write* operations; in this context, a *read* is simply a *read&write* that does not alter the value.

Note that the relationship between the VSC-write/VL-write problems and the VSC/VL problems with read&write only is analogous to the relationship between the VSC-read/VL-read problems and the basic VSC/VL problems, namely, the distinction between providing the identity of the previous write versus providing only the value of the previous write.

VSC with read&write only. We show that the VSC problem with read&write only is NP-complete, even under two restrictive scenarios.

By Theorem 4.3, the VSC problem restricted to instances in which each location is written to by only a single processor is NP-complete. In such instances, each write can be replaced with the appropriate read&write operation, yielding the following corollary.

COROLLARY 4.7. *The VSC problem with read&write only restricted to instances in which each location is written to by only a single processor is* NP*-complete.*

We next observe that instances with long processor sequences can always be transformed to equivalent instances with at most two memory operations per processor.

OBSERVATION 4.8. *There is a linear-time reduction from the VSC problem with read&write only with p sequences, n operations, and k variables to the VSC problem with read&write only with at most n sequences, $O(n)$ operations, and k variables such that each sequence contains at most two operations.*

Proof. For $i = 1, 2, \ldots, p$, consider the ith sequence in the instance, $S_i = rw(a_1, d_1 : d'_1)$, $rw(a_2, d_2 : d'_2)$, ..., $rw(a_{m_i}, d_{m_i} : d'_{m_i})$, where $a_1, \ldots, a_{m_i}$, $d_1, \ldots, d_{m_i}$, $d'_1, \ldots, d'_{m_i}$ are not necessarily distinct. When S_i has more than two memory operations (i.e., $m_i > 2$), the construction splits each operation $rw(a_j, d_j : d'_j)$ in S_i, other than the first and the last operation, into a pair of operations to the same address: $rw(a_j, d_j : x^{(i)}_{j-1})$ and $rw(a_j, x^{(i)}_{j-1} : d'_j)$. In particular, we replace S_i with the following $m_i - 1$ replacement sequences for S_i:

$$\begin{array}{cccccc} rw(a_1, d_1 : d'_1) & rw(a_2, x^{(i)}_1 : d'_2) & rw(a_3, x^{(i)}_2 : d'_3) & \cdots & rw(a_{m_i-1}, x^{(i)}_{m_i-2} : d'_{m_i-1}) \\ rw(a_2, d_2 : x^{(i)}_1) & rw(a_3, d_3 : x^{(i)}_2) & rw(a_4, d_4 : x^{(i)}_3) & & rw(a_{m_i}, d_{m_i} : d'_{m_i}) \end{array},$$

where $\forall i, j, i', j'$, $x^{(i)}_j = x^{(i')}_{j'}$ if and only if $i = i'$ and $j = j'$.

The idea behind this construction is as follows. Consider the pairs of operations in the constructed instance. Since each new data value is unique, then in any legal schedule of the constructed instance, no operation to the same address can be scheduled between a pair, and no operation from the same replacement sequences can be scheduled between a pair. This in turn will imply that any legal schedule can be reordered to obtain a new legal schedule in which the two operations in any pair are consecutive. Then considering each pair as being replaced by its original operation, we have a legal schedule of the original instance.

The reader may verify that this constructed instance is a positive instance if and only if the original instance is a positive instance. □

We now show that the VSC problem with read&write only is NP-complete even when there is only a single variable and at most two operations per sequence.

THEOREM 4.9. *The VSC problem with read&write only restricted to instances with one variable and at most two memory operations per sequence is* NP*-complete.*

Proof. We show the reduction for sequences with many operations; this can be transformed to sequences with at most two operations each by applying Observation 4.8. Given a 3SAT instance $\mathcal{F}$ with n variables and m clauses, we construct the following VSC instance $\mathcal{V}$. First, we have the following $2n + 1$ sequences, A, $V_1, V'_1, \ldots, V_n, V'_n$, where $\perp$ is the initial value of a, the literal v_1 is in clauses $\{c_2, c_3, \ldots, c_m\}$, the literal $\overline{v}_1$ is in clauses $\{c_4, c_6, \ldots, c_9\}$,..., the literal v_n is in

clauses $\{c_4, c_5, \ldots, c_8\}$, and the literal $\overline{v}_n$ is in clauses $\{c_1, c_6, \ldots, c_m\}$:

$\underline{A}$	$\underline{V_1}$	$\underline{V_1'}$	$\cdots$	$\underline{V_n}$	$\underline{V_n'}$
$rw(a, \perp : 1)$	$rw(a, 1 : c_2)$	$rw(a, 1 : c_4)$		$rw(a, n : c_4)$	$rw(a, n : c_1)$
$rw(a, 1' : 2)$	$rw(a, c_2 : c_3)$	$rw(a, c_4 : c_6)$		$rw(a, c_4 : c_5)$	$rw(a, c_1 : c_6)$
$rw(a, 2' : 3)$	:	:		:	:
:	$rw(a, c_m : 1')$	$rw(a, c_9 : 1')$		$rw(a, c_8 : n')$	$rw(a, c_m : n')$
$rw(a, (n-1)' : n)$					

In addition, we have the following $m + 1$ sequences, A', $C_1, C_2, \ldots, C_m$:

$\underline{A'}$	$\underline{C_1}$	$\underline{C_2}$	$\cdots$	$\underline{C_m}$
$rw(a, c_m' : 1)$	$rw(a, c_1 : c_1)$	$rw(a, c_2 : c_2)$		$rw(a, c_m : c_m)$
$rw(a, 1' : 2)$	$rw(a, n' : c_1')$	$rw(a, c_1' : c_2')$		$rw(a, c_{m-1}' : c_m')$
$rw(a, 2' : 3)$				
:				
$rw(a, (n-1)' : n)$				

LEMMA 4.10. *Let $\mathcal{F}$ be an instance of the* 3SAT *problem, and let $\mathcal{V}$ be the instance of the VSC problem with read&write only constructed as defined above. Then $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable. Let $T(v_1), \ldots, T(v_n)$ be a satisfying assignment for $\mathcal{F}$, where $T(v_i) \in \{\mathsf{T}, \mathsf{F}\}$. We construct the following schedule for $\mathcal{V}$:

1. first, $rw(a, \perp : 1)$ (or $rw(\perp : 1)$; from now on we shall omit the variable "a"); this is the first operation in A;
2. then if $T(v_1) = \mathsf{T}$, all of the operations in V_1, interleaved with *clause read* operations, as explained below, and followed by the second operation in A, $rw(1' : 2)$; otherwise, if $T(v_1) = \mathsf{F}$, all of the operations in V_1', interleaved with *clause read* operations, as explained below, and followed by the second operation in A.
3. For $i = 2, 3, \ldots, n$, repeat the previous step: If $T(v_i) = \mathsf{T}$ ($T(v_i) = \mathsf{F}$), schedule all of the operations in V_i (respectively, V_i'), interleaved with *clause read* operations, as explained below, and followed by, for $i < n$, the $(i+1)$th operation in A, $rw(i' : i + 1)$.
4. The first operation in each sequence C_j is denoted the *clause read* for clause c_j. Since T satisfies $\mathcal{F}$, then each clause c_j is satisfied by some $T(v_i)$, and therefore when the corresponding V_i or V_i' was scheduled, an operation of the form $rw(x : c_j)$ for some value x was in that sequence; the clause read for c_j is scheduled immediately after the first such operation $rw(x : c_j)$.
5. Next, we schedule $rw(n' : c_1')$, $rw(c_1' : c_2'), \ldots, rw(c_{m-1}' : c_m')$, followed by $rw(c_{m'} : 1)$ (the first operation in A').
6. Finally, repeat the following for $i = 1, 2, \ldots, n$: If $T(v_i) = \mathsf{F}$ ($T(v_i) = \mathsf{T}$), schedule all of the operations in V_i (respectively, V_i'), followed by, for $i < n$, the $(i+1)$th operation in A': $rw(i' : i + 1)$.

The reader may verify that this is a legal schedule.

Conversely, suppose $\mathcal{V}$ is a positive instance and let S be a legal schedule for $\mathcal{V}$. For $i = 1, \ldots, n$, let the first operation in S from either V_i or V_i' be denoted the *variable read* operation for v_i and the sequence V_i or V_i' containing v_i be denoted the *variable read sequence*. Let the sequence V_i or V_i' that is not the variable read sequence for v_i be denoted the *cleanup read sequence* and its first operation be denoted the *cleanup read*. Consider the truth assignment T defined as follows: For each $i = 1, \ldots, n$,

$T(v_i) = \mathsf{T}$ if V_i is the variable read sequence for v_i and $T(v_i) = \mathsf{F}$ otherwise. Note that by the construction, operations in a variable read sequence only write values c_j for clauses satisfied by T. We will show that T is a satisfying assignment for $\mathcal{F}$.

Suppose there is a clause c_j not satisfied by T, and consider the clause read for c_j: $rw(c_j : c_j)$. Let $S = \sigma_1 rw(c_j : c_j)\sigma_2$. We claim that $rw(c'_m : 1)$ and hence all of A' is in σ_2. To see this, first observe that since the values $c'_1, c'_2, \ldots, c'_m$ are read by exactly one operation and written by exactly one operation, the sequence $rw(n' : c'_1)$, $rw(c'_1 : c'_2), \ldots, rw(c'_{m-1} : c'_m)$, $rw(c'_m : 1)$ is a consecutive subsequence of S. Since the second operation in C_j is in this subsequence and in σ_2, then $rw(c'_m : 1)$ is in σ_2 as well. Thus all of A' is in σ_2, and hence at most one operation that writes i, $i = 1, \ldots, n$, is in σ_1.

Since among the two operations that read i, the variable read precedes the cleanup read, it follows that the cleanup read and hence the cleanup read sequence is in σ_2 for all v_i. The last operation in σ_1 must write c_j and thus must be in a variable read sequence. Thus, as observed above, c_j is satisfied by T, a contradiction.

The lemma follows. □

Since the above transformation can be done in polynomial time, Theorem 4.9 follows. □

VL with read&write only. We show that the VL problem with read&write only is NP-complete. This contrasts with the $O(n \log n)$-time algorithm for the VL-write problem.

THEOREM 4.11. *The VL problem with read&write only is* NP*-complete.*

Proof. With only read&write operations, a more careful construction is needed than the one shown in Figure 8 that relies on writes whose old values are not predetermined. Each new value to be written must serve as the old value for the next write (recall that there is but a single location), giving us less flexibility in the construction. Nevertheless, we show below how to overcome this difficulty to obtain a construction with the desired properties.

Our reduction is again from SAT. Consider an instance $\mathcal{F}$ of SAT with n variables, $v_1, v_2, \ldots, v_n$, and m clauses, $C_1, C_2, \ldots, C_m$. Without loss of generality, assume that each variable and its negation appear in at least one clause, but not the same clause, and that there are no repeated variables in a clause. We construct an instance of the VL problem with at most $5nm + 10n + m + 1$ operations, corresponding to $\mathcal{F}$. To simplify the description, we have multiple intervals with common start times or end times; these ties can be broken arbitrarily to ensure unique time values.

Figure 10 depicts an example construction. First, there is an initial read&write operation: *read&write*$(a, \perp : 0, 1, 2)$. Then for each clause C_j, $j = 1, \ldots, m$, we have a *read&write*$(a, c_j : c_j, 3, 5n + 4)$ operation, denoted the *clause read* for C_j.

For $i = 1, \ldots, n$, let m_i (respectively, $m_{\bar{i}}$) be the number of clauses containing the literal v_i (respectively, $\overline{v_i}$). By assumption, $m_i > 0$, $m_{\bar{i}} > 0$, and $m_i + m_{\bar{i}} \leq m$. For $i = 1, \ldots, n+1$, let $\Delta_i = (7m+4)(i-1)+5n+4$. For $i = 1, \ldots, n$, let $\Delta_{\bar{i}} = \Delta_i + 7m_i + 2$.

For each variable v_i, assignment to v_i is simulated using six operations:

- *read&write*$(a, 0 : i, 5i, 5i + 1)$,
- *read&write*$(a, 0 : \bar{i}, 5i, 5i + 1)$,
- *read&write*$(a, \hat{i} : 0, 5i, 5i + 1)$,
- *read&write*$(a, \hat{i} : 0, 5i + 3, 5i + 4)$,
- *read&write*$(a, i : \hat{i}, 5i, \Delta_i - 1)$,
- *read&write*$(a, \bar{i} : \hat{i}, 5i, \Delta_{\bar{i}} - 1)$.

There is a set of operations for each literal v_i, denoted the *group* of operations

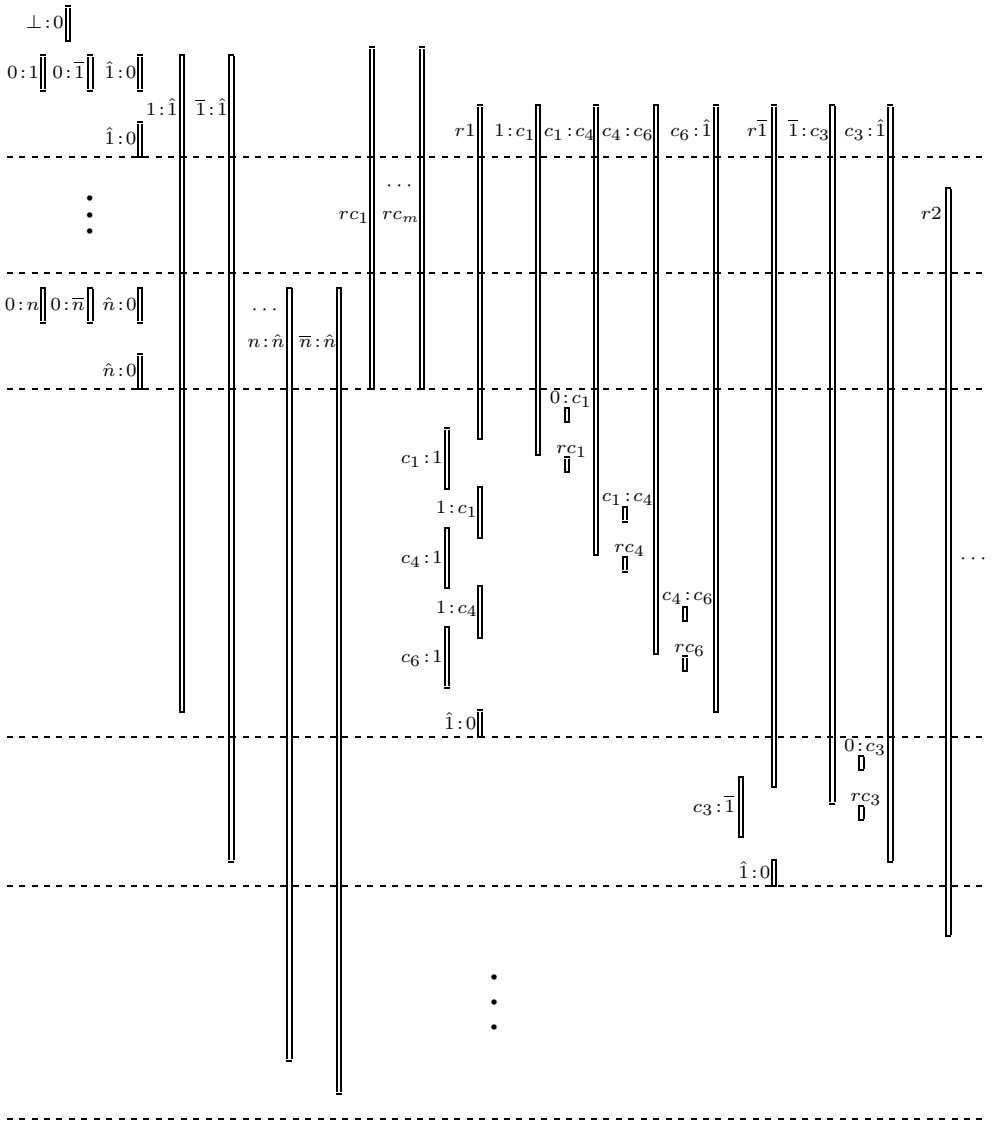

FIG. 10. *Transforming an instance of* SAT *to an instance of VL with a single location such that all memory operations are read&write operations. There are n variables, $v_1, \ldots, v_n$, and m clauses, $C_1, \ldots, C_m$. The full construction has at most $5nm + 10n + m + 1$ operations. Here the literal v_1 appears in exactly clauses C_1, C_4, and C_6, the literal $\overline{v}_1$ appears in clause C_3 only, and so forth. Every column corresponds to a processor. Vertical boxes depict the intervals of time for the respective read&write operations to the single location; time progresses from top to bottom in the figure. A read&write operation with old value d and new value d' is denoted $d:d'$; the case where $d = d'$ is denoted simply rd. The set of values used is $\{0, 1, \overline{1}, \hat{1}, 2, \overline{2}, \hat{2}, \ldots, n, \overline{n}, \hat{n}, c_1, c_2, \ldots, c_m\}$. The first read&write necessarily reads the initial value in memory, which we denote $\perp$, before writing a new value (i.e., 0).*

for i. For $i = 1, \ldots, n$, if $C_{i_1}, C_{i_2}, \ldots, C_{i_{m_i}}$ are the clauses containing the literal v_i, we have the following $5m_i + 2$ intervals. First, for C_{i_1}, we have

- *read&write*$(a, i\!:\!i, 5i+2, \Delta_i+4)$,
- *read&write*$(a, i\!:\!c_{i_1}, 5i+2, \Delta_i+5)$,
- *read&write*$(a, 0\!:\!c_{i_1}, \Delta_i+1, \Delta_i+2)$,
- *read&write*$(a, c_{i_1}\!:\!c_{i_1}, \Delta_i+5, \Delta_i+6)$,
- *read&write*$(a, c_{i_1}\!:\!i, \Delta_i+3, \Delta_i+7)$.

Then for C_{i_k}, $k = 2, \ldots, m_i$, we have

- *read&write*$(a, i\!:\!c_{i_{k-1}}, \Delta_i+7(k-1), \Delta_i+7(k-1)+4)$,
- *read&write*$(a, c_{i_{k-1}}\!:\!c_{i_k}, 5i+2, \Delta_i+7(k-1)+5)$,
- *read&write*$(a, c_{i_{k-1}}\!:\!c_{i_k}, \Delta_i+7(k-1)+1, \Delta_i+7(k-1)+2)$,
- *read&write*$(a, c_{i_k}\!:\!c_{i_k}, \Delta_i+7(k-1)+5, \Delta_i+7(k-1)+6)$,
- *read&write*$(a, c_{i_k}\!:\!i, \Delta_i+7(k-1)+3, \Delta_i+7k)$.

Finally, we have *read&write*$(a, c_{i_{m_i}}\!:\!\hat{\imath}, 5i+2, \Delta_{\bar{\imath}}-1)$ and *read&write*$(a, \hat{\imath}\!:\!0, \Delta_{\bar{\imath}}-1, \Delta_{\bar{\imath}})$.

Likewise, there is a set of operations for each literal $\overline{v_i}$, denoted the *group* of operations for $\bar{\imath}$. For $i = 1, \ldots, n$, if $C_{\bar{i}_1}, C_{\bar{i}_2}, \ldots, C_{\bar{i}_{m_{\bar{i}}}}$ are the clauses containing the literal $\overline{v_i}$, we have the $5m_{\bar{\imath}} + 2$ intervals obtained from the previous definition by replacing $\Delta_{\bar{\imath}}$ with Δ_{i+1} and leaving "$5i+2$" unchanged but otherwise replacing i with $\bar{\imath}$ throughout.

This completes the construction.

We show below that for any satisfying truth assignment, there is a legal schedule for the instance constructed. As an example, consider a satisfying assignment that sets v_1 to true, and refer to Figure 10. In this case, a legal schedule begins: $\perp : 0$, then $0 : \overline{1}$, then $\overline{1} : \hat{1}$, then the first $\hat{1} : 0$, then $0 : 1$, then $r1$, then $1 : c_1$ (immediately to the right of $r1$ in the figure), then the clause read rc_1, then $c_1 : c_4$, then the clause read rc_4 (not shown), then $c_4 : c_6$, then the clause read rc_6 (not shown), then $c_6 : \hat{1}$, and then the second $\hat{1} : 0$. The schedule continues with operations for $v_2, v_3, \ldots, v_n$ until the second $\hat{n} : 0$ is scheduled. Then the remaining (cleanup) operations for v_1 are scheduled as follows: $0 : c_1$ (in the figure, just below the third dashed line from the top), then rc_1, then $c_1 : 1$, then the $1 : c_1$ to its right, then the $c_1 : c_4$ to its right, then the rc_4 below it, then $c_4 : 1$, then $1 : c_4$, then the $c_4 : c_6$ to its right, then the rc_6 below it, then $c_6 : 1$, then $1 : \hat{1}$, and then the $\hat{1} : 0$ just above the fourth dashed line; then $0 : c_3$, then $c_3 : \overline{1}$, then $r\overline{1}$, then $\overline{1} : c_3$, then rc_3, then $c_3 : \hat{1}$, and then the $\hat{1} : 0$ just above the fifth dashed line. The schedule continues with cleanup operations for $v_2, v_3, \ldots, v_n$ until all operations have been scheduled.

LEMMA 4.12. *Let $\mathcal{F}$ be an instance of a* SAT *problem, and let $\mathcal{V}$ be the instance of the VL problem with read&write operations constructed as described above. Then $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable, and let $\mathcal{T}$ be a satisfying truth assignment for $\mathcal{F}$. We construct the following schedule for $\mathcal{V}$. When there is no ambiguity, we use the notation "$d_1 : d_2$" to denote a read&write operation to address a with old value d_1 and new value d_2; the case where $d_1 = d_2$ is denoted simply "rd_1." We show the order in which events are scheduled; an assignment of distinct, valid times to operations is left to the reader. The schedule is as follows:

1. First, schedule $\perp : 0$.
2. Repeat the following for $i = 1, 2, \ldots, n$:
   If v_i is set to true by $\mathcal{T}$, schedule $0 : \bar{\imath}$, then $\bar{\imath} : \hat{\imath}$, then *read&write*$(a, \hat{\imath} : 0, 5i, 5i+1)$, then $0 : i$, and then ri. Schedule $i : c_{i_1}$, followed by—if it has not already been scheduled—the clause read rc_j, where $j = i_1$. Then repeat

for $k = 2, \ldots, m_i$: schedule *read&write*$(a, c_{i_{k-1}} : c_{i_k}, 5i + 2, \Delta_i + 7(k-1) + 5)$, followed by—if it has not already been scheduled—the clause read rc_j, where $j = i_k$. Finally, schedule $c_{i_{m_i}} : \hat{\imath}$, then *read&write*$(a, \hat{\imath} : 0, 5i + 3, 5i + 4)$.

The case where v_i is set to false by $\mathcal{T}$ is symmetric and left to the reader.

3. Since $\mathcal{T}$ is a satisfying assignment for $\mathcal{F}$, all clause reads have been scheduled by this point. Note also that all the above operations can be scheduled prior to time Δ_1.

   Repeat the following for $i = 1, 2, \ldots, n$:

   If v_i is set to true by $\mathcal{T}$, consider the $4m_i$ unscheduled operations in group i, together with the unscheduled $i : \hat{\imath}$, and finally the $5m_{\bar{\imath}} + 2$ (unscheduled) operations in group $\bar{\imath}$:

   (a) Schedule $0 : c_{i_1}$, then rc_{i_1}, then $c_{i_1} : i$.

   (b) Then repeat for $k = 2, \ldots, m_i$: Schedule $i : c_{i_{k-1}}$, then the unscheduled $c_{i_{k-1}} : c_{i_k}$, then rc_{i_k}, and then $c_{i_k} : i$.

   (c) Schedule the unscheduled $i : \hat{\imath}$, then schedule *read&write*$(a, \hat{\imath} : 0, \Delta_{\bar{\imath}} - 1, \Delta_{\bar{\imath}})$ to complete group i.

   (d) Next, schedule $0 : c_{\bar{\imath}_1}$, then $c_{\bar{\imath}_1} : \bar{\imath}$, then $r\bar{\imath}$, then $\bar{\imath} : c_{\bar{\imath}_1}$, and then $rc_{\bar{\imath}_1}$.

   (e) Then repeat for $k = 2, \ldots, m_{\bar{\imath}}$: Schedule *read&write*$(a, c_{\bar{\imath}_{k-1}} : c_{\bar{\imath}_k}, \Delta_{\bar{\imath}} + 7(k-1) + 1, \Delta_{\bar{\imath}} + 7(k-1) + 2)$, then $c_{\bar{\imath}_k} : \bar{\imath}$, then $\bar{\imath} : c_{\bar{\imath}_{k-1}}$, then the unscheduled $c_{\bar{\imath}_{k-1}} : c_{\bar{\imath}_k}$, and then $rc_{\bar{\imath}_k}$.

   (f) Finally, to complete group $\bar{\imath}$, schedule $c_{\bar{\imath}_{m_{\bar{\imath}}}} : \hat{\imath}$ and then *read&write*$(a, \hat{\imath} : 0, \Delta_{i+1} - 1, \Delta_{i+1})$.

   The case where v_i is set to false by $\mathcal{T}$ is symmetric and left to the reader.

The reader may verify that this is a legal schedule.

The proof of the converse, i.e., if $\mathcal{V}$ is a positive instance, then $\mathcal{F}$ is satisfiable, parallels the proof given in Lemma 3.6 and is left to the reader. □

Since the above transformation can be done in polynomial time, Theorem 4.11 follows. □

4.4. Providing the conflict-order. Two operations on the same location *conflict* if at least one is a write. In the *VSC-conflict* and *VL-conflict* problems, both a read-mapping and a write-order are known, implying that all conflicting operations are ordered; a legal schedule must respect this *conflict-order* relation.

VSC-conflict. There is a simple $O(n \log n)$-time algorithm for the VSC-conflict problem. Thus although providing either the read-mapping or the write-order is NP-complete, providing both yields a fast algorithm.

THEOREM 4.13. *There is an $O(n \log n)$-time algorithm for the VSC-conflict problem.*

Proof. Sort the VSC-conflict instance by address. We check to see that each read is mapped to a write operation with the same address and value and that each write is included only in the write-order for its address; otherwise, we have a negative instance. Construct a graph G with a vertex for each operation in the instance. Add an edge between a vertex u and a vertex v if u immediately precedes v in some processor sequence. In addition, add edges from each write to all reads mapped to it and from the write and these reads to the next write in its write-order. The graph G has n vertices and $O(n)$ edges.

The reader may verify that if G has a directed cycle, we have a negative instance. Otherwise, we have a positive instance, and any topological sort of G yields a legal schedule. □

VL-conflict. An $O(n \log n)$-time algorithm for the VL-conflict problem follows as a corollary to Theorems 3.2 and 4.13.

COROLLARY 4.14. *There is an $O(n \log n)$-time algorithm for the VL-conflict problem.*

4.5. Atomic read–modify–write. We conclude this section by showing how the algorithmic results of this paper can be extended to handle atomic read–modify–write operations at no asymptotic penalty. The input consists of three types of memory operations: reads, writes, and read&writes. The read&write operations are included in both the domain and range of any read-mapping, as well as ordered by any write-order (in which case the value read must match the value written by the previous write).

Few processors. We first consider the scenario where the number of processors is bounded.

THEOREM 4.15. *The VL problem with read, write, and read&write operations, restricted to instances such that at any time t, there are at most k operations on the same address whose intervals contain t, can be solved in $O(n2^{O(k)} + n \log n)$ time.*

Proof. The proof parallels the proof of Theorem 3.7, with the following modification of the algorithm to handle the possible presence of read&write operations. Namely, condition 2 for including an edge between a vertex v and a vertex v' on consecutive levels of the graph defines a set A of operations that must be assigned times after the last write ω designated by v and before any writes not in A. With only reads and writes, it sufficed to test whether each read in A read either the value of the last write designated by v or the value of some write in A. With read&write operations, however, we cannot have, for example, the set A consist of two operations that each read the designated last write and write a new value. We use the following test instead. Consider a multigraph H consisting of a vertex for every value occurring in $A \cup \{\omega\}$, plus an extra dummy vertex z. Each read&write operation in A that reads a value d and writes a value d' defines a directed edge in H from the vertex for d to the vertex for d'. Each write operation defines a directed edge from z to the vertex for the value written. Each read operation defines a self-loop at the vertex for the value read. There is a directed edge from z to the vertex for the value written by ω. Then for each vertex y in H whose in-degree exceeds its out-degree by $j > 0$, we have j directed dummy edges in H from y to z.

LEMMA 4.16. *The set of operations in A can be safely scheduled if and only if H has an Eulerian circuit.*

Proof. If H has an Eulerian circuit, then consider the sequence S of operations defined by an Eulerian circuit that begins with the edge for ω. Then the schedule obtained from S by removing ω and all dummy edges contains all of A and has no reads-from violations. Conversely, if there is no Eulerian circuit, then either H has a vertex x whose out-degree exceeds its in-degree or it has a nonempty set of vertices not reachable from z. In the former case, note that $x \neq z$ since by construction the in-degree of z is necessarily at least as large as its out-degree. Moreover, since any vertex with an outgoing dummy edge has the same in-degree as out-degree, all edges incident to x correspond to operations in $A \cup \{\omega\}$. Hence there are more operations that read and modify x's value than there are operations that write it. It follows that any schedule of A will have a reads-from violation. In the latter case, any schedule of A will have a reads-from violation at the first scheduled operation from the unreachable set. □

We construct H and check to see that all vertices are reachable from z and have the same in-degree as out-degree in time linear in the size of A. Finally, consider the last write, ω', designated by v', which like ω may be either a write or a read&write. The final condition required for adding an edge from v to v' is that either ω' is in A and writes a value whose vertex in H has an outgoing dummy edge or, if there are no writes or read&writes in A, $\omega' = \omega$. □

Theorem 4.15 implies, for instance, the following corollary.

COROLLARY 4.17. *There is an $O(n \log n)$-time algorithm for the VL problem with read, write, and read&write operations and any fixed number of processors.*

Read-mapping. If a read-mapping is provided, then we have the following corollary to Theorem 4.2.

COROLLARY 4.18. *There is an $O(n \log n)$-time algorithm for the VL-read problem with read, write, and read&write operations.*

Proof. We modify the algorithm in Theorem 4.2 to handle read&write operations by simply replacing the use of *clusters* in that algorithm with *cluster sequences*, as defined below. Observe that at most one read&write operation can be mapped by the read-mapping to a single write or read&write; otherwise, we have a negative instance. Define a *cluster sequence*, $S = w_1, R_1, w_2, R_2, \ldots, w_k, R_k$, $k \geq 1$, to be a sequence of alternating operations and sets such that w_1 is an ordinary write operation, $w_2, \ldots, w_k$ are read&write operations with w_i mapped to w_{i-1} for $i = 2, \ldots, k$, there is no read&write operation mapped to w_k, and R_i is the set of ordinary reads mapped to w_i for $i = 1, \ldots, k$. Any legal schedule assigns w_1 an earlier time than any read in R_1, which is assigned an earlier time than w_2, and so on. Thus the start-of-interval time for an operation must be earlier than the end-of-interval time for any operation that is ordered after it in its cluster sequence; otherwise, we have a negative instance. A legal assignment of times defines an interval for a cluster sequence, from the time assigned to w_1 to the time assigned to the last read in R_k; only operations in this cluster sequence can be scheduled during this time interval. The algorithm and proof continue as in Theorem 4.2. □

Write-order and conflict-order. Since the old value on writes provides no additional information once a write-order is provided, we have the following corollary to Theorems 4.6 and 4.13 and Corollary 4.14.

COROLLARY 4.19. *There are $O(n \log n)$-time algorithms for the VL-write, VSC-conflict, and VL-conflict problems with read, write, and read&write operations.*

5. Conclusions. This paper provides the first formal and systematic study of the complexity of testing the correctness of an execution of a shared memory, based on the reads and writes observed by the individual processors. We define two combinatorial problems: the *verifying sequential consistency of shared-memory executions* (VSC) problem for testing for sequential consistency, and the *verifying linearizability of shared-memory executions* (VL) problem for testing for linearizability. We show that the VSC problem is NP-complete even when the number of processors, locations, or operations per processor is bounded. Moreover, the problem remains NP-complete even when additional information is provided, such as the pairing of reads to particular writes, the order of writes to a location, or the old value overwritten by each write. However, if the order of all conflicting operations to a location is provided, we obtain an $O(n \log n)$-time algorithm. Linearizability, in contrast, is more restrictive than sequential consistency, and our results show that these additional restrictions can be quite useful in testing for linearizability. In particular, we present $O(n \log n)$-time algorithms for several variants of the VL problem whose corresponding VSC variants

are NP-complete. On the other hand, we show that the VL problem is NP-complete even when given the old value overwritten by each write operation.

Efficient testing algorithms can be used by machine designers, system software writers, application programmers, and naïve users whenever doubt arises as to whether all aspects of the memory system are functioning properly. In [17, 19], we present approaches to implementing testing procedures on real multiprocessors or their simulators. In particular, we devise schemes for collecting the additional information needed for the read-mapping, write-order, old value on writes, or conflict-order. We also describe algorithms for heuristic testing of several processors at a time. The results in [17, 19] demonstrate interesting tradeoffs between the speed, accuracy, and obtrusiveness of various testing procedures.

We have presented several $O(n \log n)$-time algorithms; after sorting, these run in linear time. Depending on the assumptions made on the form of the input, one could apply integer sorting to obtain linear time bounds for these algorithms.

In this paper, we consider read, write, and read–modify–write operations. This is extended in [17, 19], where we also consider the load-reserved (a.k.a. load-linked, load-locked) and store-conditional operations [24] appearing in many recent architectures. More generally, one can consider testing shared memories that support various data structures, such as priority queues.

Finally, correctness conditions other than linearizability and sequential consistency can be considered. A number of correctness conditions from the domain of database transactions have been previously studied (e.g., [6, 25, 30]). It would be interesting to develop a general theory encompassing a wide range of correctness conditions so that tradeoffs between the generality of a correctness condition and its complexity may be better understood.

Acknowledgments. We thank Michael Merritt and Robert Cypher for discussions related to this work. We acknowledge the helpful comments of an anonymous referee.

REFERENCES

[1] S. V. Adve, *Designing memory consistency models for shared-memory multiprocessors*, Ph.D. thesis, University of Wisconsin, Madison, WI, 1993.

[2] 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 Symposium on Computer Architecture, ACM, New York, 1991, pp. 234–243.

[3] Y. Afek, G. M. Brown, and M. Merritt, *Lazy caching*, ACM Trans. Programming Lang. Systems, 15 (1993), pp. 182–205.

[4] Y. Afek, D. S. Greenberg, M. Merritt, and G. Taubenfeld, *Computing with faulty shared objects*, J. Assoc. Comput. Mach., 42 (1995), pp. 1231–1274.

[5] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung, *The MIT Alewife machine: A large-scale distributed-memory multiprocessor*, in Proc. Workshop on Scalable Shared Memory Multiprocessors, Kluwer Academic Publishers, Norwell, MA, 1991.

[6] D. Agrawal, J. L. Bruno, A. El Abbadi, and V. Krishnaswamy, *Relative serializability: An approach for relaxing the atomicity of transactions*, in Proc. 13th ACM Symposium on Principles of Database Systems, ACM, New York, 1994, pp. 139–149.

[7] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, *The Tera computer system*, in Proc. 1990 International Conference on Supercomputing, ACM, New York, 1990, pp. 1–6.

[8] H. Attiya and J. L. Welch, *Sequential consistency versus linearizability*, ACM Trans. Comput. Systems, 12 (1994), pp. 91–122.

[9] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, *Checking the correctness of memories*, Algorithmica, 12 (1994), pp. 225–244.

[10] J. Bright and G. Sullivan, *Checking mergeable priority queues*, in Proc. 24th IEEE Fault-Tolerant Computing Symposium, IEEE, Piscataway, NJ, 1994, pp. 144–153.
[11] W. W. Collier, *Reasoning About Parallel Architectures*, Prentice–Hall, Englewood Cliffs, NJ, 1992.
[12] S. Frank, H. Burkhardt III, and J. Rothnie, *The KSR1: Bridging the gap between shared memory and MPPs*, in Proc. 1993 IEEE Compcon Spring, IEEE, Piscataway, NJ, 1993, pp. 285–294.
[13] 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.
[14] K. Gharachorloo and P. B. Gibbons, *Detecting violations of sequential consistency*, in Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, 1991, pp. 316–326.
[15] K. Gharachorloo, A. Gupta, and J. Hennessy, *Two techniques to enhance the performance of memory consistency models*, in Proc. 1991 International Conference on Parallel Processing, CRC Press, Boston, MA, 1991, pp. I:355–364.
[16] 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 Symposium on Computer Architecture, IEEE, Piscataway, NJ, 1990, pp. 15–26.
[17] P. B. Gibbons and E. Korach, *Testing for sequential consistency*, in preparation.
[18] P. B. Gibbons and E. Korach, *The complexity of sequential consistency*, in Proc. 4th IEEE Symposium on Parallel and Distributed Processing, IEEE, Piscataway, NJ, 1992, pp. 317–325.
[19] P. B. Gibbons and E. Korach, *On testing cache-coherent shared memories*, in Proc. 6th ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, 1994, pp. 177–188.
[20] P. B. Gibbons and M. Merritt, *Specifying nonblocking shared memories*, in Proc. 4th ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, 1992, pp. 306–315.
[21] P. B. Gibbons, M. Merritt, and K. Gharachorloo, *Proving sequential consistency of high-performance shared memories*, in Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, 1991, pp. 292–303.
[22] M. P. Herlihy and J. M. Wing, *Linearizability: A correctness condition for concurrent objects*, ACM Trans. Programming Lang. Systems, 12 (1990), pp. 463–492.
[23] R. Hood, K. Kennedy, and J. Mellor-Crummey, *Parallel program debugging with on-the-fly anomaly detection*, in Proc. 1990 International Conference on Supercomputing, ACM, New York, 1990, pp. 74–81.
[24] E. H. Jensen, G. W. Hagensen, and J. M. Broughton, *A new approach to exclusive data access in shared memory multiprocessors*, Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, CA, 1987.
[25] V. Krishnaswamy and J. Bruno, *On the complexity of concurrency control using semantic information*, Technical Report TRCS 92-21, Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA, 1992.
[26] L. Lamport, *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Trans. Comput., C-28 (1979), pp. 690–691.
[27] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, *The Stanford DASH multiprocessor*, IEEE Comput., 25 (1992), pp. 63–79.
[28] J. Mellor-Crummey, *Compile-time support for efficient data race detection in shared-memory parallel programs*, in Proc. 3rd ACM/ONR Workshop on Parallel and Distributed Debugging, ACM, New York, 1993, pp. 129–139.
[29] R. H. B. Netzer and B. P. Miller, *Improving the accuracy of data race detection*, in Proc. 3rd ACM Symposium on Principles and Practice of Parallel Programming, ACM, New York, 1991, pp. 133–144.
[30] C. Papadimitriou, *The serializability of concurrent database updates*, J. Assoc. Comput. Mach., 26 (1979), pp. 631–653.
[31] C. Papadimitriou, *The Theory of Database Concurrency Control*, Computer Science Press, Rockville, MD, 1986.
[32] F. Pong and M. Dubois, *A new approach for the verification of cache coherence protocols*, IEEE Trans. Parallel Distrib. Systems, 6 (1995), pp. 773–787.
[33] D. Shasha and M. Snir, *Efficient and correct execution of parallel programs that share memory*, ACM Trans. Programming Lang. Systems, 10 (1988), pp. 282–312.
[34] J. M. Wing and C. Gong, *Testing and verifying concurrent objects*, J. Parallel Distrib. Comput., 17 (1993), pp. 164–182.