

The Complexity of Verifying Memory Coherence

by

JASON F. CANTIN

Abstract

Memory coherence is an important feature of shared memory multiprocessor systems. In this work, we study the problem of verifying memory coherence for multiprocessor executions. We prove that determining whether memory coherence was maintained for an execution is NP-Complete given the memory operations performed and their associated data values.

Verifying memory coherence remains NP-Complete when each process is allowed only three memory operations and each data value is written at most twice. Similar results hold when only read-modify-write operations are allowed. The problem is tractable when restricted to a constant number of processes, data values written only once, or when given the order in which write operations were performed.

The NP-Completeness of verifying memory coherence implies the NP-Hardness of verifying adherence to a memory consistency model that requires coherence, such as sequential consistency. Furthermore, our results extend to consistency models that do not strictly require coherence, but allow the programmer to serialize memory operations with special synchronization instructions. However, the complexity of verifying adherence to a consistency model is not a mere consequence of these results. We also show that verifying sequential consistency remains NP-Complete for executions known to be coherent, or for which coherence may be verified tractably.

Finally, we outline methods for verifying coherence in the general case. These include a recursive algorithm with some simple heuristics, and a transformation to the well-known problem SAT. Though neither method yields a deterministic polynomial-time algorithm, cases requiring exponential time seem unlikely to occur in practice.

Acknowledgements

This research was supported by NSF grant CCR-0083126, IBM, and fellowships from the NSF and the University of Wisconsin Foundation. It is a privilege to acknowledge James E. Smith and Mikko H. Lipasti for their guidance during the course of this research. Thanks to Deborah Joseph, Eric Bach, and Dieter van Melkebeek for discussions related to this work. This work also benefited from the many comments and suggestions of Paramjit Oberoi, Shiliang Hu, Kevin Lepak, Trey Cain, and Carl Mauer.

Table of Contents

Abstract.....	ii
Acknowledgements	iii
Table of Contents	iv
Table of Figures.....	vi
1 Introduction.....	1
2 Memory Coherence and Consistency	3
2.1 Memory Coherence	3
2.2 Memory Consistency.....	6
3 Related Work	8
4 Preliminaries	10
4.1 Notation	10
4.2 Definitions.....	11
5 Complexity of Verifying Memory Coherence	15
5.1 Simple Reads and Writes.....	15
5.2 Read-Modify-Write Operations	24
5.3 General Case	32
6 Complexity of Restricted and Augmented Cases.....	34
6.1 Restricting Number of Memory Operations	34
6.2 Restricting Number of Processes	38
6.3 Restricting Data Value Locality.....	39
6.4 Supplying the Order of Writes	43
6.5 Combining Restrictions	45
6.6 Summary of Restricted Cases	47
7 Beyond Coherence: Complexity of Verifying Consistency	49
7.1 Sequential Consistency.....	49
7.2 Other Memory Consistency Models.....	50
7.3 Complexity of Consistency with Coherence.....	52
8 Closing The Loop.....	60

8.1	Simple Algorithm.....	60
8.2	Optimizing the Simple Algorithm.....	61
8.3	Reduction to SAT.....	62
9	Future Work.....	67
10	Conclusions.....	68
11	References.....	69
	Appendix.....	72
A.1	Complexity and NP.....	72
A.2	Decision Problems and Promise Problems.....	73
A.3	Proving NP-Completeness.....	74
A.4	Propositional Satisfiability.....	75

Table of Figures

Figure 2.1: Memory Coherence Example.....	5
Figure 2.2: Conceptual View of Sequential Consistency.	6
Figure 5.1: Reduction of SAT to VMC.	18
Figure 5.2: Example SAT Instance and Corresponding Instance of VMC.	19
Figure 5.3: Previously Shown Instance of VMC with Coherent Schedule.	20
Figure 5.4: Unsatisfiable Instance of SAT and Corresponding Instance of VMC.....	21
Figure 5.5: Previously Shown Instance of VMC with Incomplete Schedule.	22
Figure 5.6: Reduction of SAT to VMC-RMW.	28
Figure 5.7: Example SAT Instance and Corresponding Instance of VMC-RMW.....	29
Figure 5.8: Previously Shown Instance of VMC-RMW with Coherent Schedule.....	30
Figure 6.1: Reduction of SAT to VMC, Only Three Memory Operations Per Process....	35
Figure 6.2: Reduction of 3SAT to VMC-RMW with Only Two Operations Per Process.	36
Figure 6.3: Reduction of 3SAT to VMC with Values Written At Most Twice.	41
Figure 6.4: Reduction of 3SAT to VMC-RMW, Values Written at Most Three Times. ...	42
Figure 6.5: 3SAT to VMC, Three Operations Per Process and Values Written Twice. ...	46
Figure 6.6: 3SAT to VMC-RMW, Two Operations and Values Written Three Times. ...	47
Figure 6.7: Summary of Complexity Results for VMC.	48
Figure 7.1: SAT to VMC with Synchronization for Each Operation.....	51
Figure 7.2: Reduction of SAT to VSCC.....	55
Figure 7.3: VSCC with Memory Operations Separated by Address.....	58
Figure 8.1: SAT Instance Generated From Example VMC Instance.....	65
Figure 8.2: Optimized SAT Instance Generated From Example VMC Instance.....	66

1 Introduction

An important feature in a shared-memory multiprocessor system is *memory coherence*. Memory coherence provides software with the illusion that there is a single copy of each shared variable, and operations from the different processors are performed one at a time. To maintain this illusion a shared-memory system must ensure that, for each physical location, the memory operations from all the processes appeared to execute in a serial order. Specifically, a sequence in which memory operations from the same process appear in the order specified by the program, and read operations return the value of the immediately preceding write operation in the sequence.

However, it is becoming increasingly difficult to design shared-memory multiprocessor systems that maintain memory coherence. First, the design complexity of these systems is increasing to offer competitive performance. Modern systems incorporate write buffers, levels of cache, unordered networks, and various forms of speculation to improve performance. Consequently, it is becoming extremely difficult to ensure that a design will provide coherence for all executions. Second, shrinking transistor dimensions and rising power dissipation in integrated circuits are increasing the susceptibility of hardware to runtime errors. Third, traditional approaches to fault-tolerance have focused only on the detection of data corruption and computation errors [1,2]. There is a lack of work addressing the problem of detecting violations of coherence.

For these reasons and others, we are researching techniques for detecting violations of memory coherence at runtime. This paper presents a theoretical investigation into the complexity of the problem that was carried out as a first step in this research. First, verifying memory coherence is stated formally as a decision problem (VMC), and proven NP-Complete when only the operations and data values are available. In other words, for a given address, if we are given all the memory operations performed by each process and the corresponding data values, finding a serial order in which each read returns the value of the immediately preceding write is an NP-Complete problem. The problem is then analyzed under a number of restrictions. The problem remains NP-Complete for as few as three memory operations per process, or at most two writes of each data value.

The problem is also NP-Complete for as few as two read-modify-write operations per process, or values written by at most three read-modify-write operations.

The complexity results obtained for verifying memory coherence are then extended to reason about the complexity of verifying adherence to memory consistency models. The NP-Completeness of verifying memory coherence directly implies the NP-Hardness of verifying adherence to memory consistency models that require coherence, including sequential consistency. Furthermore, our results extend to consistency models that do not strictly require coherence, but allow the programmer to force a serialization with synchronization instructions. However, the complexity of verifying adherence to a memory consistency model is not a mere consequence of requiring memory coherence. We show that verifying sequential consistency remains NP-Complete for executions that are known to be coherent, and for which coherence may be verified tractably.

Finally, we propose methods for verifying coherence in the general case. First, a recursive algorithm with some simple heuristics is presented. Though the algorithm's run-time grows exponentially with the number of processes and memory operations in the worst-case, such cases seem unlikely to occur in practice. Second, we show how the problem of verifying coherence can be converted into the well-known problem SAT, for which well-developed algorithms exist.

2 Memory Coherence and Consistency

Coherence and consistency are correctness conditions that define how a shared memory must behave from the point of view of software. Coherence requires that memory operations with the same address appear to execute in some serial order, as if performed one at a time. Consistency places constraints on the ordering of all memory operations, and typically subsumes coherence. These properties form an interface between software and the memory system that is intuitive and simple to understand.

2.1 Memory Coherence

In his 1976 paper, Tang observed that *cache-transparency* would be violated if caches were integrated with the individual processors in a multiprocessor system [7]. Unless the caches communicated to keep cached copies of the data updated, the caches would become visible to software and violate the abstraction of processors sharing access to a single copy of memory. Later, this was dubbed the *cache coherence problem*, and it was stated “the value returned on a LOAD instruction is always the value given by the latest STORE instruction with the same address” [8].

Implicit in this original definition is a notion of physical time, and the assumption that processors execute operations in program order (they typically did at the time). However, sophisticated optimizations have since complicated the picture, and rendered correctness conditions based on notions of physical time inadequate. Data may be buffered and replicated in a variety of places besides the cache, such that the term *cache coherence* is now somewhat misleading. We adopt *memory coherence* as a more appropriate term [9].

Memory coherence has a variety of definitions in the literature [8,9,10,11,12,13,14]. Some definitions are implicit, or functional in nature, giving a set of sufficient conditions for an implementation [9,11]. Others list invariants for a particular protocol to establish [10,14]. Different architectures specify different requirements for the

behavior of the memory system in multiprocessor configurations [15,16]. Therefore, it is important to explicitly define the term.

In this work, we use the following definition for memory coherence, to be stated formally in the next section:

An execution on a shared-memory system is *coherent for a given location* if the memory operations from all processes, in their respective program orders, can be interleaved into a schedule in which the data returned by each read is that of the immediately preceding write. A shared-memory execution is *coherent* if it is coherent for each shared location.

This definition is essentially write-synchronization, as defined by Collier [13]. There are no constraints on the ordering between memory operations to different locations (such constraints usually fall under the jurisdiction of consistency).

The phrase “every shared location” warrants some elaboration. We assume here that all memory operations are aligned accesses of a single word for simplicity. Without this assumption, our definition must allow for a collection of writes preceding a read when a read overlaps the data of more than one write. To consider misaligned accesses, we must define whether they should appear atomic (since they affect multiple locations, their atomicity can imply a stronger ordering than required here).

In contrast to other definitions, we avoid specifying properties like bounded write-propagation (written data eventually becomes visible to other processes), fairness in arbitration, and forward progress [8,9,10,11,14]. We cannot verify these properties without knowledge of the particular implementation, or some notion of physical time. We also avoid defining coherence in terms of the interactions between different processors [10]. Data is logically shared between threads of control, regardless of the processor.

As an example, examine the sequences of memory operations in Figure 2.1 (all the memory operations have the same address). The first set is coherent, with a serial order indicated. The second set is not coherent, because there is no serial order in which reads return the values of the immediately preceding write. Based on the values returned

by the reads, it appears that either the writes in the first sequence or the reads in the second sequence executed out of order.

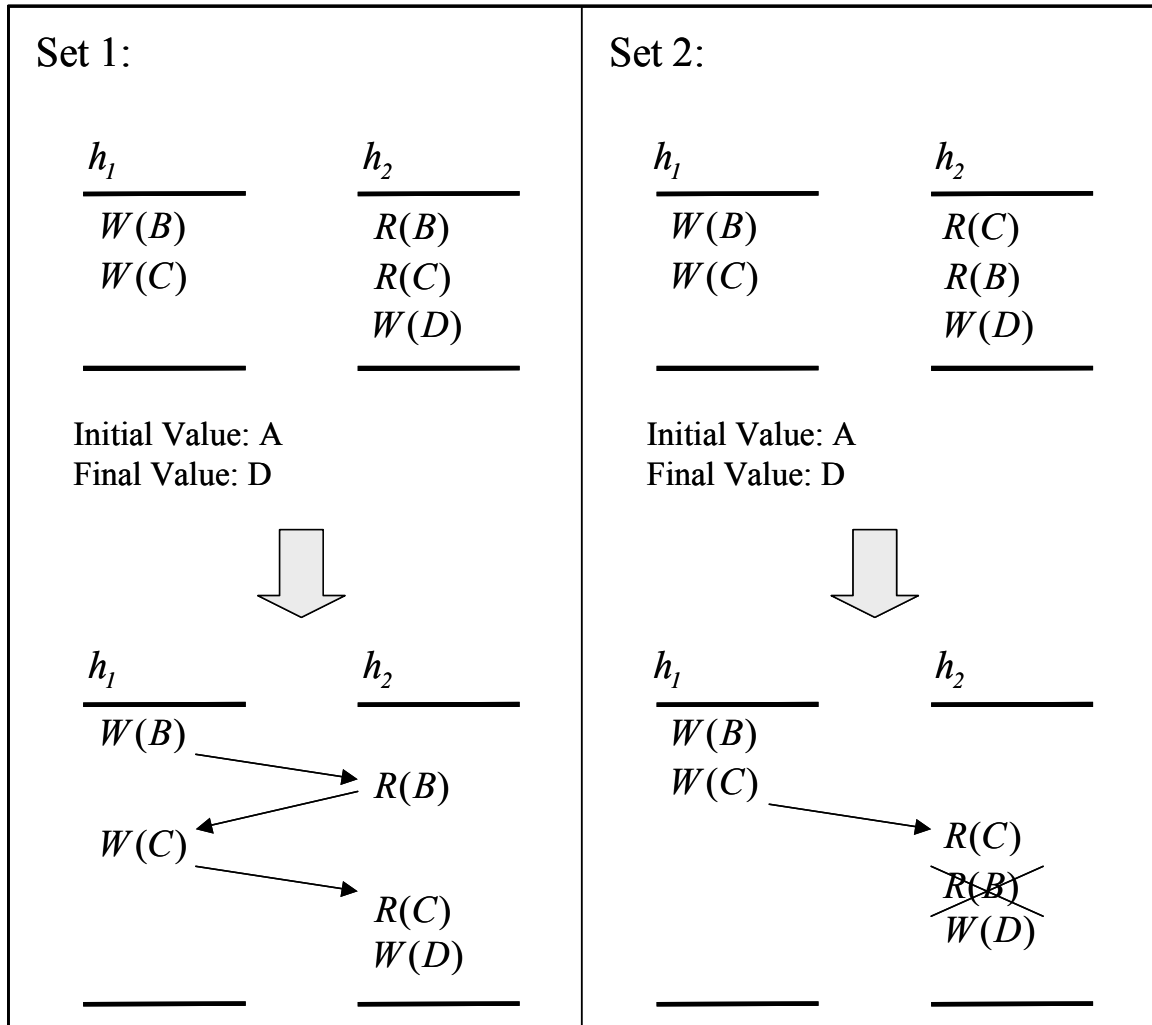


Figure 2.1: Memory Coherence Example.

Memory operations appear in program order, from top to bottom. All memory operations have the same address, with data values enclosed in parentheses. The left side (Set 1) shows an execution that is coherent, with a schedule indicated below it. The right side (Set 2) shows an execution that is not coherent.

2.2 Memory Consistency

A *memory consistency model* is a set of rules that defines how *all* memory operations should be ordered. While coherence is usually limited to memory operations that access a single location, *consistency* has additional constraints that define the ordering of memory operations with different addresses.

There are many consistency models, the most common of which is *sequential consistency* (SC), originally defined by Lamport:

“...[T]he result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program” [17].

Sequential consistency provides the illusion that all memory operations are performed in some serial order, as if the memory system executed one operation at a time. This is illustrated with a conceptual model in Figure 2.2 below (adapted from [14]). In this conceptual model, we have a memory system, a set of processors, and a moving switch that randomly selects which processor may perform the next memory operation.

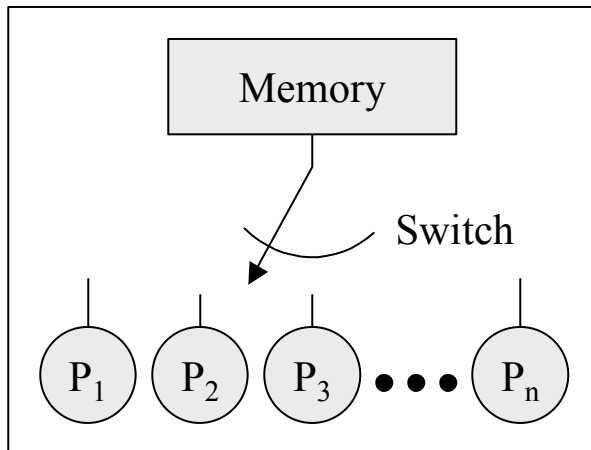


Figure 2.2: Conceptual View of Sequential Consistency.

Many other consistency models exist, most of which are weaker than SC. These include total store order (TSO), partial store order (PSO), processor consistency (PC), release consistency (RC), relaxed memory ordering (RMO), and weak ordering (WO). A more complete list with references can be found in [14]. The interested reader should consult [11,14,18] for more information about the subtleties of each model.

Each of these weaker models enables optimizations by relaxing certain ordering and atomicity constraints between memory operations. To enforce ordering constraints not implicit in the model, software must use special instructions (e.g., barriers, lock acquires and releases).

3 Related Work

The most relevant work in this area was that of Gibbons and Korach on analyzing the complexity of detecting violations of sequential consistency and linearizability [3,4,5,6]. They defined the VSC (Verifying Sequential Consistency) and VL (Verifying Linearizability) problems, and showed they were NP-Complete. They presented a collection of complexity results for restricted cases, characterizing the complexity of the problems. Having expended much effort to find an efficient method for detecting violations of memory coherence, this inspired us to determine if verifying memory coherence was in fact NP-Complete.

Our interpretation of memory coherence is equivalent to sequential consistency restricted to one location, so some of the results of Gibbons and Korach also apply to memory coherence. However, the complexity of verifying sequential consistency for one location (coherence) was not fully explored, and the general case was left as an open problem in [6]. The results contained herein address these problems, and provide new insight into the complexity of verifying memory consistency models.

Alur, McMillan, and Peled studied the model-checking problem for sequential consistency, and proved that it is in general undecidable [19]. That is, the problem of determining if a memory system implementation will provide sequential consistency for all executions. Condon and Hu extended this work in [20], identifying a restricted class of protocols for which the problem is decidable. Unlike our work and that of Gibbons and Korach, which focuses on the results of a single execution, this is the broader and more difficult problem of design verification.

Papadimitriou studied concurrency control for database transactions, and proved that verifying view-serializability is NP-Complete [21]. Similar to sequential consistency, view-serializability requires that transactions appear serialized (where a transaction is a set of operations meant to execute atomically). In contrast, the input to the decision problem is a schedule of the operations, and the task is to show that the schedule yields the same results (for all inputs) as a serial schedule.

Taylor studied synchronization in concurrent programs using the *rendezvous* mechanism [22]. That is, determining if it is possible for two tasks to synchronize at a particular point in each (via an *entry call* and an *accept statement*). Though more relevant to software verification, this also has similarities to our problem. An analogous question might be “Is it possible for a read of a particular value to be mapped to a particular write from another process?” However, a subtle difference is that not all shared-memory write operations in an execution need to be observed for coherence, whereas *accept* statements in Ada are blocking.

There has been considerable work in the area of multiprocessor job scheduling, including precedence-constrained scheduling problems. Similar to determining the order in which to execute jobs (some dependant) on a limited processing resource, a coherent schedule must be found for shared-memory operations based on the values returned. However, many write operations might write the value needed by a read, but a precedence relation only holds for one in a schedule. This work is summarized in [23].

4 Preliminaries

For precision and convenience in the proofs that follow, we introduce notation and formal definitions for reasoning about coherence and consistency. Where possible, we use symbols similar to related work [14].

4.1 Notation

Memory reads are denoted “R(a, d)”, and memory writes are denoted “W(a, d)”. Here, a represents the address, and d is the data read/written by the operation. We typically use the shorthand notation “R(d)”, and “W(d)”, since we primarily deal with operations to the same address.

In some places, atomic read-modify-write operations are considered. These will be denoted “RW(d_r , d_w)”. Unlike the simple read and write operations, these operations require two values, where the first value d_r is the data read and the second value d_w is the data written.

A memory operation Op_i may be ordered before a memory operation Op_j by the program, if both operations belong to the same process. This *program order* relation is a strict ordering, denoted as follows.

$$Op_i \xrightarrow{po} Op_j$$

A memory operation Op_i is *dynamically ordered* before a memory operation Op_j in a sequence S , if Op_i precedes Op_j in S , but Op_i and Op_j belong to different processes. This is also a strict order relation.

$$Op_i \xrightarrow{S-dyn} Op_j$$

Two memory operations are simply *ordered* if one appears before the other in a sequence S , whether or not they are from the same process. This could be a combination of dynamic and program order relations.

$$Op_i \xrightarrow{S} Op_j$$

4.2 Definitions

Prior to program execution, every location in a shared memory is in some state. Since a program may read these *initial values*, we must account for them when reasoning about coherence and consistency. Similarly, every memory location is in some state after a program has executed, and we refer to this as the *final value*.

DEFINITION 4.1: Initial Value

For each address a , $d_I[a]$ represents the initial state of the location

DEFINITION 4.2: Final Value

For each address a , $d_F[a]$ represents the final state of the location

A *history* in this context is a sequence of memory operations from the execution of a single process, in the order specified by the program. Included with each memory operation are the computed physical address, and the data read or written during the execution. Here, memory operations are listed in program order from left to right, or top to bottom.

DEFINITION 4.3: History

A history h is a sequence of memory operations from a single process, in program order

$$h \equiv \langle Op_1, Op_2, \dots, Op_n \rangle$$

$$\forall i \in [1, n) : h[i] \xrightarrow{po} h[i+1]$$

A *schedule* is a sequence of memory operations from a set of processes in which the program order of each process is preserved. In other words, a schedule is an *interleaving* of a set of histories.

DEFINITION 4.4: Schedule S for Set of Histories H

$SCHEDULE(S, H) \Leftrightarrow$

$$\forall h \forall i \left((h \in H \wedge i \in [1, |h|]) \Leftrightarrow \exists k (k \in [1, |S|] : h[i] = S[k]) \right) \wedge$$

$$\forall j \left(j \in [1, |h|] \wedge (h[i] \xrightarrow{po} h[j] \Rightarrow h[i] \xrightarrow{S} h[j]) \right)$$

A *coherent schedule* is a schedule of single-address histories, where every read operation returns the value written by the previous write operation in the schedule, except reads of the initial value. The last write in the schedule must write the final value for the location. These requirements are expressed in formal terms below (Definitions 4.5-4.7). These are used as components in our formal definition of a coherent schedule (Definition 4.8).

DEFINITION 4.5: Initial Value Requirement

$INITIAL(S, a, d_I[a]) \Leftrightarrow$

$$\exists k \forall j \left(\begin{array}{l} k \in [1, |S|] \wedge S[k] = R(a, d) \wedge \\ j \in [1, |S|] \wedge S[j] = W(a, *) \wedge (k < j) \end{array} \right) \Rightarrow d = d_I[a]$$

DEFINITION 4.6: Final Value Requirement

$FINAL(S, a, d_I[a], d_F[a]) \Leftrightarrow$

$$\left(\left(\neg \exists i (i \in [1, |S|] \wedge S[i] = W(a, d_F[a])) \wedge d_I[a] = d_F[a] \right) \vee \right. \\ \left. \left(\forall j (j \in [1, |S|] \wedge S[j] = W(a, d) \wedge d \neq d_F[a]) \Rightarrow \exists k (k \in [1, |S|] \wedge (k > j) \wedge S[k] = W(a, d_F[a])) \right) \right)$$

DEFINITION 4.7: Reads-From Requirement

$$\begin{aligned}
 & \text{READS_FROM}(S, a, d_I[a]) \Leftrightarrow \\
 & \forall i \left((i \in [1, |S|] \wedge S[i] = R(a, d) \wedge d \neq d_I[a]) \Rightarrow (S[i-1] = W(a, d) \vee S[i-1] = R(a, d)) \right)
 \end{aligned}$$

DEFINITION 4.8: Coherence Schedule S for H

$$\begin{aligned}
 & \text{COHERENT_SCHEDULE}(S, H, a, d_I[a], d_F[a]) \Leftrightarrow \\
 & \left(\begin{aligned}
 & \text{SCHEDULE}(S, H) \wedge \text{INITIAL}(S, a, d_I[a]) \wedge \\
 & \text{FINAL}(S, a, d_I[a], d_F[a]) \wedge \text{READS_FROM}(S, a, d_I[a])
 \end{aligned} \right)
 \end{aligned}$$

A set of histories from a multiprocess execution is considered coherent if, for each address there is a coherent schedule. Note that each address is considered independently, and a single schedule that is simultaneously coherent for all addresses might not exist.

DEFINITION 4.9: Coherence for a Set of Histories H

$$\text{COHERENT}(H, d_I, d_F) \Leftrightarrow \forall a \exists S (\text{COHERENT_SCHEDULE}(S, H, a, d_I[a], d_F[a]))$$

A *consistent schedule* is a schedule of all memory operations, where every read operation returns the value written by the immediately preceding write operation with the same address. In other words, the histories are sequentially consistent. As with coherent schedules, reads that precede the first write with the same address must return the initial value of the memory location, and the last write of each location must write the final value of that location.

DEFINITION 4.10: Consistent Schedule S for Set of Histories H

$$\begin{aligned}
 & \text{CONSISTENT_SCHEDULE}(S, H, d_I, d_F) \Leftrightarrow \\
 & \forall a (\text{COHERENT_SCHEDULE}(S, H, a, d_I[a], d_F[a]))
 \end{aligned}$$

A set of histories from a multiprocess execution is considered sequentially consistent if there is a consistent schedule.

DEFINITION 4.11: Sequential Consistency for a Set of Histories H

$$\exists S(\text{CONSISTENT_SCHEDULE}(S, H, d_I, d_F))$$

Note that this definition of sequential consistency implies coherence for the execution, but the converse is not necessarily true.

$$\begin{aligned} \exists S(\text{CONSISTENT_SCHEDULE}(S, H, d_I, d_F)) &\Leftrightarrow \\ \exists S \forall a(\text{COHERENT}(S, H, a, d_I[a], d_F[a])) &\Rightarrow \\ \forall a \exists S(\text{COHERENT}(S, H, a, d_I[a], d_F[a])) & \end{aligned} \quad (4.1)$$

Informally, we say that the memory operations in a history may be *scheduled* when it is possible to interleave them with an existing coherent schedule. If a new history contains a read of a value not previously written in the same history (that is not the initial value), somewhere in the existing schedule there must be a write operation with the same value. If a history contains two such reads (of different values), the history may be scheduled only if the existing schedule contains two write operations of the corresponding values in the same order.

5 Complexity of Verifying Memory Coherence

In this section, we analyze the complexity of verifying memory coherence for a shared-memory multiprocessor system. Here, we restrict ourselves to the offline problem, in which we are given a complete list of the memory operations executed (to the same address). In general, this problem is NP-Complete.

5.1 Simple Reads and Writes

We start with the simplest version of the problem, with simple read/write operations and a single memory location. That is, all memory operations have the same address, reads return the entire contents of the location, writes completely overwrite the contents of the location, and reads have no side effects.

We first state the problem of verifying memory coherence in the form of a *decision problem* (Appendix A.2). We then prove it is NP-Complete with a polynomial-time reduction from the known NP-Complete problem of Propositional Satisfiability (SAT, [23]).

DEFINITION 5.1: Verifying Memory Coherence (VMC) problem

INSTANCE: Data value set D , finite set H of process histories, each consisting of a finite sequence of read and write operations with the same address.

QUESTION: Is there a coherent schedule S for H ?

Given an instance Q of SAT in conjunctive normal form with set U of m variables and collection C of n clauses, we can construct a corresponding instance V of VMC with set H of $2m+3$ process histories and $O(mn)$ memory operations. The instance V is constructed such that a coherent schedule S exists for H if and only if there is a truth assignment T for U that satisfies C .

First, create a set D of $2m+n+2$ unique data values as shown in equation (5.1) below. There should be one element d_l for each literal over U and one element d_c for each clause in C , and two elements for the initial and final values of the memory location.

$$D \equiv \left\{ d_{u_1}, \dots, d_{u_m}, d_{\bar{u}_1}, \dots, d_{\bar{u}_m}, d_{c_1}, \dots, d_{c_n}, d_I, d_F \right\} \quad (5.1)$$

Next, create two process histories, h_1 and h_2 . For each u in U , append a write operation $W(d_u)$ to h_1 and a write operation $W(d_{\bar{u}})$ to h_2 .

$$\begin{aligned} h_1 &\equiv \langle W(d_{u_1}), W(d_{u_2}), \dots, W(d_{u_m}) \rangle \\ h_2 &\equiv \langle W(d_{\bar{u}_1}), W(d_{\bar{u}_2}), \dots, W(d_{\bar{u}_m}) \rangle \end{aligned} \quad (5.2)$$

Note that with no data dependences between h_1 and h_2 , all 2^m combinations of partial orders between the pairs of write operations are possible in a schedule S . These partial orders will encode the truth assignment T for U :

$$\begin{aligned} T : U &\mapsto \{true, false\} \\ W(d_u) &\xrightarrow{S\text{-dyn}} W(d_{\bar{u}}) \Leftrightarrow T(u) = True \\ W(d_{\bar{u}}) &\xrightarrow{S\text{-dyn}} W(d_u) \Leftrightarrow T(\bar{u}) = True \end{aligned} \quad (5.3)$$

For each literal l over U , define a subset D_l of D with elements representing each clause c in C that includes l .

$$D_l \equiv \text{An ordered list of values } d_c, \text{ such that } l \in c \quad (5.4)$$

For each variable u , create a process history h_u for the uncomplemented literal u with a read operation $R(d_u)$ followed by another read operation $R(d_{\bar{u}})$. After the read operations,

add the write operations in the subset D_u defined above (in order starting with the first). For the complementary literal \bar{u} , construct a similar history $h_{\bar{u}}$, only with the read operations in the reverse order and write operations from the subset $D_{\bar{u}}$.

$\forall u \in U :$

$$h_u \equiv \langle R(d_u), R(d_{\bar{u}}), W(D_u[1]), \dots, W(D_u[|D_u|]) \rangle \quad (5.5)$$

$$h_{\bar{u}} \equiv \langle R(d_{\bar{u}}), R(d_u), W(D_{\bar{u}}[1]), \dots, W(D_{\bar{u}}[|D_{\bar{u}}|]) \rangle$$

Next, create a process history h_3 , with read operations that can be scheduled only if all the clauses have been satisfied. For each clause c append a read operation $R(d_c)$ to h_3 .

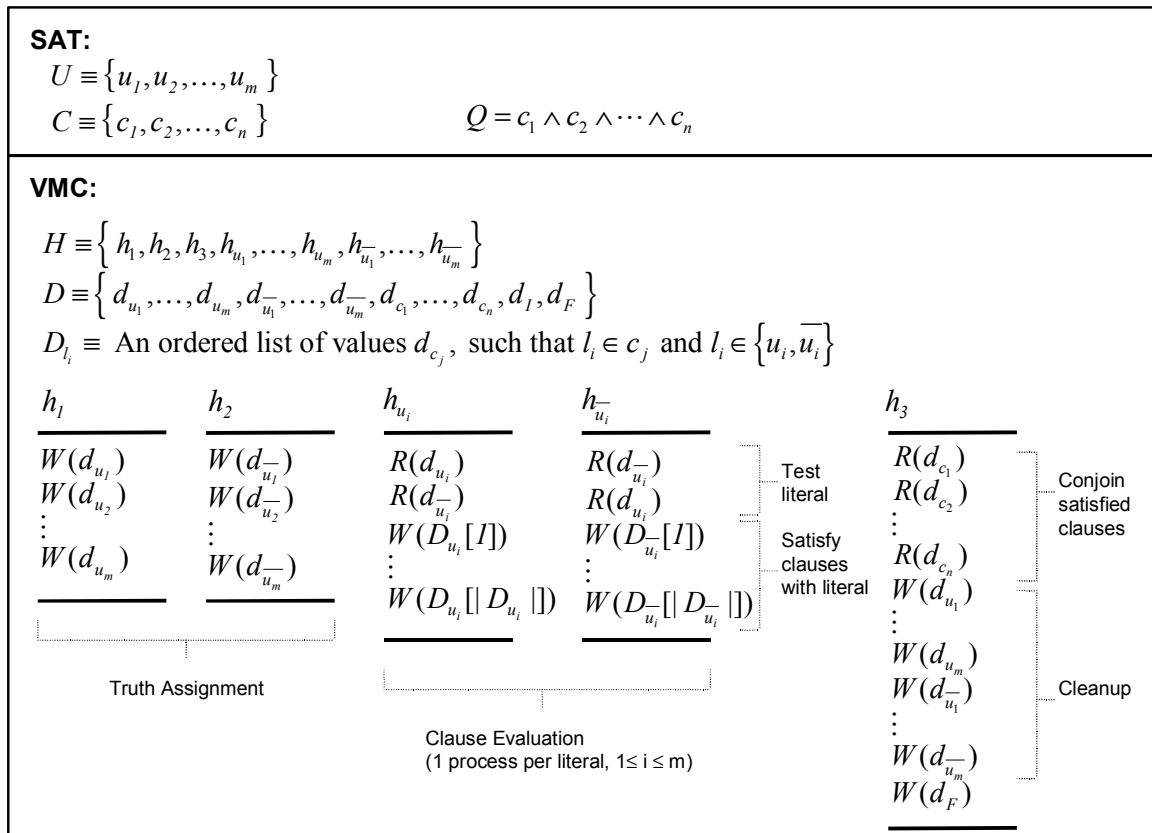
$$h_3 \equiv \langle R(d_{c_1}), R(d_{c_2}), \dots, R(d_{c_n}), \dots \rangle \quad (5.6)$$

Since only half of the literals are *true* for any truth assignment of U , we need to handle the remaining memory operations once the read operations in h_3 have been scheduled (a “cleanup” phase). Append a set of write operations $\{W(d_u), W(d_{\bar{u}})\}$ to h_3 for each u in U . Complete the history with a write of the final value, d_F .

$$h_3 \equiv \langle R(d_{c_1}), R(d_{c_2}), \dots, R(d_{c_n}), W(d_{u_1}), \dots, W(d_{u_m}), W(d_{\bar{u}_1}), \dots, W(d_{\bar{u}_m}), W(d_F) \rangle \quad (5.7)$$

The reduction is now complete; let H be the set of process histories we have just constructed. See Figure 5.1 for an annotated summary.

$$H \equiv \left\{ h_1, h_2, h_3, h_{u_1}, \dots, h_{u_m}, h_{\bar{u}_1}, \dots, h_{\bar{u}_m} \right\} \quad (5.8)$$



<p>SAT Instance:</p> $U \equiv \{u_1, u_2\}$ $C \equiv \{c_1 : \{u_1, u_2\}, c_2 : \{\bar{u}_1, u_2\}\}$ $Q = (u_1 \vee u_2) \wedge (\bar{u}_1 \vee u_2)$						
<p>VMC Instance:</p> $H \equiv \{h_1, h_2, h_3, h_{u_1}, h_{u_2}, h_{u_1}^-, h_{u_2}^-\}$ $D \equiv \{d_{u_1}, d_{u_2}, d_{u_1}^-, d_{u_2}^-, d_{c_1}, d_{c_2}, d_I, d_F\}$ $D_{u_1} \equiv \{d_{c_1}\} \quad D_{u_1}^- \equiv \{d_{c_2}\}$ $D_{u_2} \equiv \{d_{c_1}, d_{c_2}\} \quad D_{u_2}^- \equiv \emptyset$						
h_1	h_2	h_{u_1}	$h_{u_1}^-$	h_{u_2}	$h_{u_2}^-$	h_3
<u>$W(d_{u_1})$</u>	<u>$W(d_{u_1}^-)$</u>	<u>$R(d_{u_1})$</u>	<u>$R(d_{u_1}^-)$</u>	<u>$R(d_{u_2})$</u>	<u>$R(d_{u_2}^-)$</u>	<u>$R(d_{c_1})$</u>
<u>$W(d_{u_2})$</u>	<u>$W(d_{u_2}^-)$</u>	<u>$R(d_{u_1}^-)$</u>	<u>$R(d_{u_1})$</u>	<u>$R(d_{u_2}^-)$</u>	<u>$R(d_{u_2})$</u>	<u>$R(d_{c_2})$</u>
		<u>$W(d_{c_1})$</u>	<u>$W(d_{c_2})$</u>	<u>$W(d_{c_1})$</u>		<u>$W(d_{u_1})$</u>
				<u>$W(d_{c_2})$</u>		<u>$W(d_{u_2})$</u>
						<u>$W(d_{u_1}^-)$</u>
						<u>$W(d_{u_1}^-)$</u>
						<u>$W(d_{u_2}^-)$</u>
						<u>$W(d_{c_1})$</u>

Figure 5.2: Example SAT Instance and Corresponding Instance of VMC.

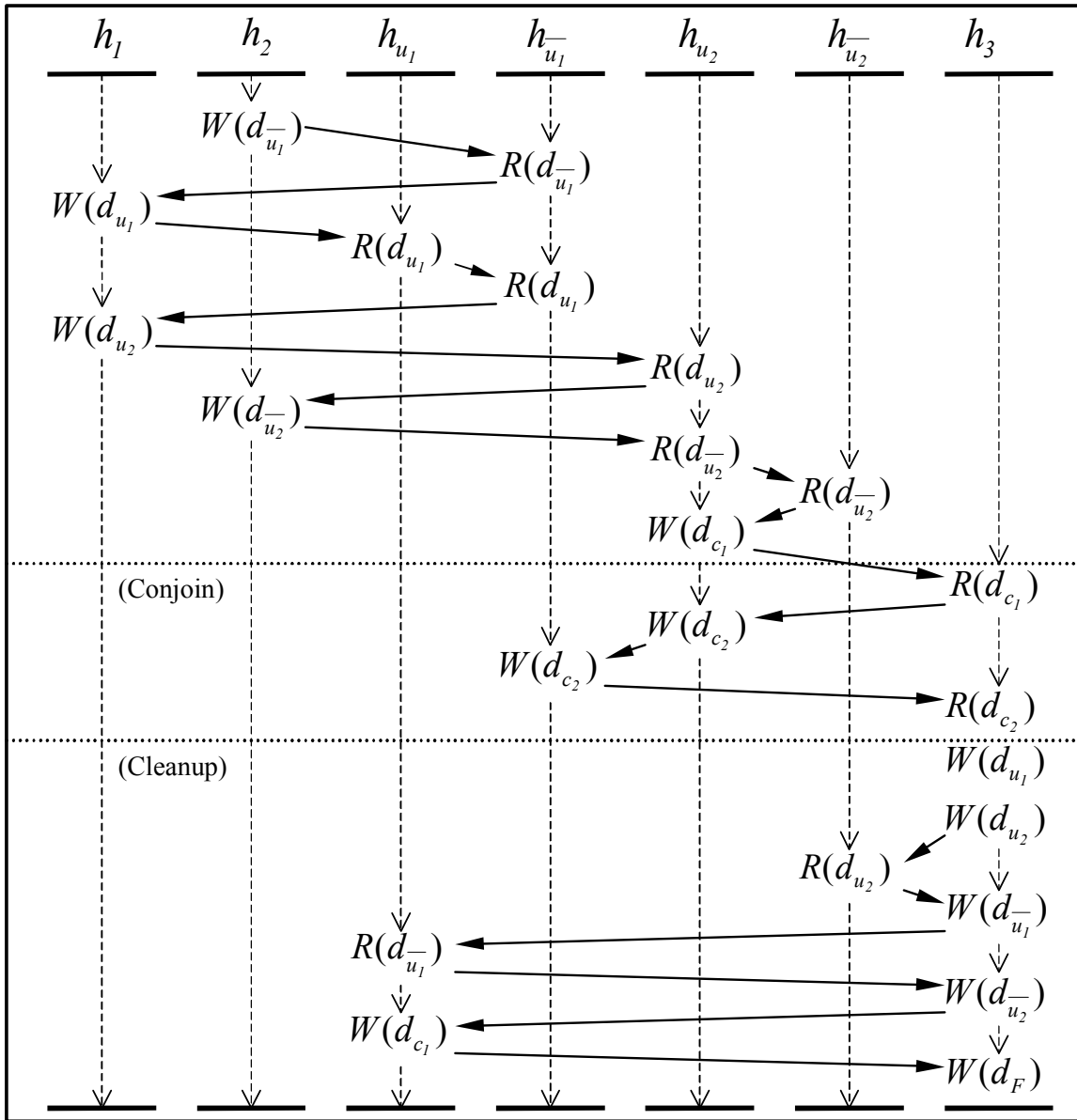


Figure 5.3: Previously Shown Instance of VMC with Coherent Schedule.

Figure 5.4 shows a very small instance that is not satisfiable, and the corresponding instance of VMC. Note that it is not possible to construct a coherent schedule for resulting process histories. In Figure 5.5, we illustrate this by choosing a truth assignment and attempting to construct a schedule.

<p>SAT Instance:</p> $U \equiv \{u_1\}$ $C \equiv \{c_1 : \{u_1\}, c_2 : \{\bar{u}_1\}\}$ $Q = u_1 \wedge \bar{u}_1$				
<p>VMC Instance:</p> $H \equiv \{h_1, h_2, h_3, h_{u_1}, h_{u_1^-}\}$ $D \equiv \{d_{u_1}, d_{u_1^-}, d_{c_1}, d_{c_2}, d_I, d_F\}$ $D_{u_1} \equiv \{d_{c_1}\} \quad D_{u_1^-} \equiv \{d_{c_2}\}$				
h_1	h_2	h_{u_1}	$h_{u_1^-}$	h_3
<hr/>	<hr/>	<hr/>	<hr/>	<hr/>
$W(d_{u_1})$	$W(d_{u_1^-})$	$R(d_{u_1})$	$R(d_{u_1^-})$	$R(d_{c_1})$
<hr/>	<hr/>	$R(d_{u_1^-})$	$R(d_{u_1})$	$R(d_{c_2})$
<hr/>	<hr/>	$W(d_{c_1})$	$W(d_{c_2})$	$W(d_{u_1})$
<hr/>	<hr/>	<hr/>	<hr/>	$W(d_{u_1^-})$
<hr/>	<hr/>	<hr/>	<hr/>	$W(d_I)$
<hr/>	<hr/>	<hr/>	<hr/>	$W(d_F)$
<hr/>	<hr/>	<hr/>	<hr/>	<hr/>

Figure 5.4: Unsatisfiable Instance of SAT and Corresponding Instance of VMC.

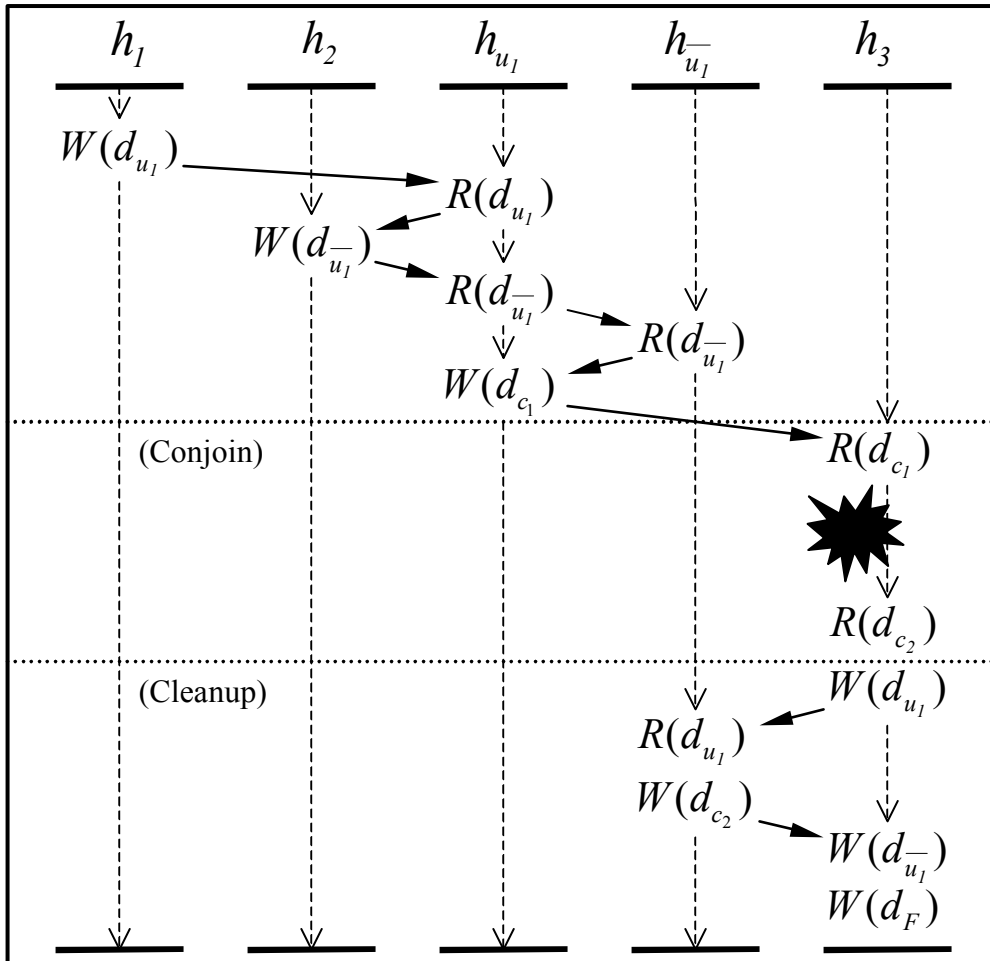


Figure 5.5: Previously Shown Instance of VMC with Incomplete Schedule.

Here, the variable has been assigned *true*. The clause c_2 is not satisfied under the assignment. The schedule cannot be completed because the corresponding read is not immediately preceded by a write of the same value.

THEOREM 5.2: VMC is NP-Complete

Proof:

1. Membership in NP:

A certificate for VMC is a schedule of all the memory operations. Given a schedule S for a set of process histories H , we can easily determine if it is a coherent schedule. We can scan S to verify that it contains all the memory operations from

H , in their respective process orders. While scanning S , we record the latest value written, and ensure that subsequent read operations return that value.

2. NP-Hardness:

Let Q be an arbitrary instance of SAT, and V the corresponding instance of VMC defined above with set D of data values and set H of process histories.

LEMMA 5.3: V is coherent if and only if Q is satisfiable.

Proof:

- 1) Suppose V is coherent. By definition, there exists a coherent schedule S for H . For each variable u , the write operations $\{W(d_u), W(d_{\bar{u}})\}$ from h_1 and h_2 appear in S in some order. By equation (5.3), the order encodes the truth assignment T for U . If $W(d_u)$ precedes $W(d_{\bar{u}})$ in S the variable u is *true* under T , and *false* otherwise.

For each clause c , there is a read operation $R(d_c)$ in h_3 that forces a write operation $W(d_c)$ to precede the write operations of h_3 in S . However, a write operation $W(d_c)$ only appears in the process histories that represent literals that appear in c . The write operations from a subset of the process histories representing the literals, in which there is at least one write $W(d_c)$ for each clause in C , must precede the writes of h_3 in S .

In order for the write operations of a process history representing a literal to precede the writes of h_3 in S , the read operations at the beginning of the history must precede the writes of h_3 in S , by program order. Besides h_3 , the only other process histories that write the values read by the process histories representing the literals are h_1 and h_2 , and these write each value only once. This means that h_1 and h_2 are interleaved in such a way in S that the corresponding literal is *true* under the assignment T . Hence, for every clause c in C , at least one of the literals in c must be *true* under T .

Because a clause c is satisfied if at least one of its literals is *true* under T , must satisfy every clause in C . Therefore, Q is satisfiable.

- 2) To prove the converse is true, suppose Q is satisfiable. There is a truth assignment T for U that satisfies every clause in C . Use T to interleave h_1 and h_2 as defined. The set of literals assigned *true* under T correspond to the set of process histories that may be interleaved with h_1 and h_2 to form a schedule S .

The process history for each literal contains a write operation $W(d_c)$ for each of the clauses c it appears in, which may precede the writes of h_3 in S if and only if the corresponding literal is *true* under T . Since T satisfies C , all clauses are satisfied, and at least one literal per clause c is *true*. Hence, at least one of the process histories containing each write operation $W(d_c)$ may precede the write operations of h_3 in S .

There is a read operation $R(d_c)$ in h_3 for every clause c before the write operations. Since a write of the same value may also precede the write operations in h_3 , a write of the same value may precede each of the reads in h_3 . Hence, it is possible to construct a schedule S that includes h_3 .

The write operations from h_3 can precede the read operations in the process histories representing literals that are *false* under T , allowing all the remaining process histories to be added to S . Therefore, V is coherent, and Lemma 5.3 follows.

Since VMC is in NP, and SAT reduces to VMC in polynomial time, VMC is NP-Complete. \square

5.2 Read-Modify-Write Operations

In the previous section, the VMC problem was proven NP-Complete for the case of simple reads and writes. However, modern machines typically include atomic memory

access instructions for implementing synchronization primitives. Most of these are some form of read-modify-write operation, where the processor atomically updates the data. While these are intended for synchronization, the fact that coherence protocols typically acquire an exclusive copy of a block before writing individual bytes allows us to treat every memory operation like a read-modify-write [6].

Based on this idea, we can define a different version of the VMC problem, in which we only have read-modify-write operations. Effectively, we know the previous value of the location for every write. Simple read operations are mimicked by read-modify-writes that write the value read. The new decision problem is as follows:

DEFINITION 5.4: VMC with Read-Modify-Writes (VMC-RMW) Problem.

INSTANCE: Value set D , finite set H of process histories, each consisting of a finite sequence of read-modify-write operations to the same address.

QUESTION: Is there a coherent schedule S for H , where all operations read the value written by the preceding operation?

Schedules are further constrained (over VMC) because all operations have data dependences between them. Every operation has a read component that depends on the previous operation in the sequence, and a write component on which the next operation depends. Regardless, we will show this problem is also NP-Complete.

Before proceeding further, it is important to note that this particular result is not new. Gibbons and Korach showed verifying sequential consistency is NP-Complete when all memory operations are read-modify-writes of the same variable [5,6]. Our definition of coherence is functionally equivalent to their definition of sequential consistency when restricted to one address.

Assume we are given an arbitrary instance Q of SAT in conjunctive normal form, with set U of m variables ($2m$ literals) and collection C of n clauses.

Create a set D of $m+n+3$ unique data values: one for each variable, one for each clause, a sentinel value, and the initial and final values. Label the elements of D as shown in equation (5.9) below.

$$D \equiv \{d_{u_1}, \dots, d_{u_m}, d_{c_1}, \dots, d_{c_n}, d_p, d_I, d_F\} \quad (5.9)$$

For each literal over U , define a subset of D with elements representing each clause c that includes the literal, as done previously for Theorem 5.2 (5.4).

For each variable u , create a pair of process histories $\{h_u, h_{\bar{u}}\}$ for the literals, beginning with a read-modify-write operation $RW(d_u, d_p)$. Next, append operations that read the sentinel value, and overwrite it with a value corresponding to each clause in which the literal appears (in order starting with the first).

$\forall u \in U :$

$$h_u \equiv \langle RW(d_u, d_p), RW(d_p, D_u[1]), \dots, RW(d_p, D_u[|D_u|]) \rangle \quad (5.10)$$

$$h_{\bar{u}} \equiv \langle RW(d_u, d_p), RW(d_p, D_{\bar{u}}[1]), \dots, RW(d_p, D_{\bar{u}}[|D_{\bar{u}}|]) \rangle$$

Note that the process history for only one of the literals of each variable u may be scheduled once the corresponding value d_u is written, because the value is atomically read and overwritten. A truth assignment T for U is encoded by the order in which the first operation of the process histories corresponding to the literals of each variable appear in a schedule S .

$T : U \mapsto \{true, false\}$

$$RW(h_u, d_u, d_p) \xrightarrow{dyn} RW(h_{\bar{u}}, d_{\bar{u}}, d_p) \Leftrightarrow T(u) = True \quad (5.11)$$

$$RW(h_{\bar{u}}, d_{\bar{u}}, d_p) \xrightarrow{dyn} RW(h_u, d_u, d_p) \Leftrightarrow T(\bar{u}) = True$$

Create a process history h_I , to act as a “scheduler” for the truth assignment. The first m read-modify-write operations write the values in D corresponding to each variable, and read d_p . Next, add read-modify-write operations that can be scheduled only if all the clauses have been satisfied. This consists of an operation $RW(d_c, d_p)$ for each c .

$$h_1 \equiv \left\langle \begin{array}{l} RW(d_p, d_{u_1}), \dots, RW(d_p, d_{u_m}), \\ RW(d_{c_1}, d_p), \dots, RW(d_{c_n}, d_p), \dots \end{array} \right\rangle \quad (5.12)$$

With read-modify-write operations, there must be a read of a value for every write of that value, except the final value. Thus, cleanup operations are needed to reset the memory location to d_p after a write of a value corresponding to a clause that has already been satisfied. If a clause c has k literals, then $RW(d_c, d_p)$ must appear k times in a sequence S . Create a process history h_c for each clause c with read-modify-write operations ($k-1$ for k literals in the clause).

$$\forall c \in C, k = |c|: \quad (5.13)$$

$$h_c \equiv \langle RW_1(d_c, d_p), \dots, RW_{k-1}(d_c, d_p) \rangle$$

To handle the remaining process histories for *false* literals, we complete the cleanup by repeating the first m read-modify-write operations at the end of the h_1 .

$$h_1 \equiv \left\langle \begin{array}{l} RW(d_I, d_{u_1}), \dots, RW(d_p, d_{u_m}), \\ RW(d_{c_1}, d_p), \dots, RW(d_{c_n}, d_p), \\ RW(d_p, d_{u_1}), \dots, RW(d_p, d_{u_m}), RW(d_p, d_F) \end{array} \right\rangle \quad (5.14)$$

The reduction is now complete; let H be the set of $(2m+n+2)$ process histories constructed above. See Figure 5.6 for an annotated summary.

$$H \equiv \left\{ h_1, h_2, h_{u_1}, \dots, h_{u_m}, h_{u_1}^-, \dots, h_{u_m}^-, \dots, h_{c_1}, \dots, h_{c_n} \right\} \quad (5.15)$$

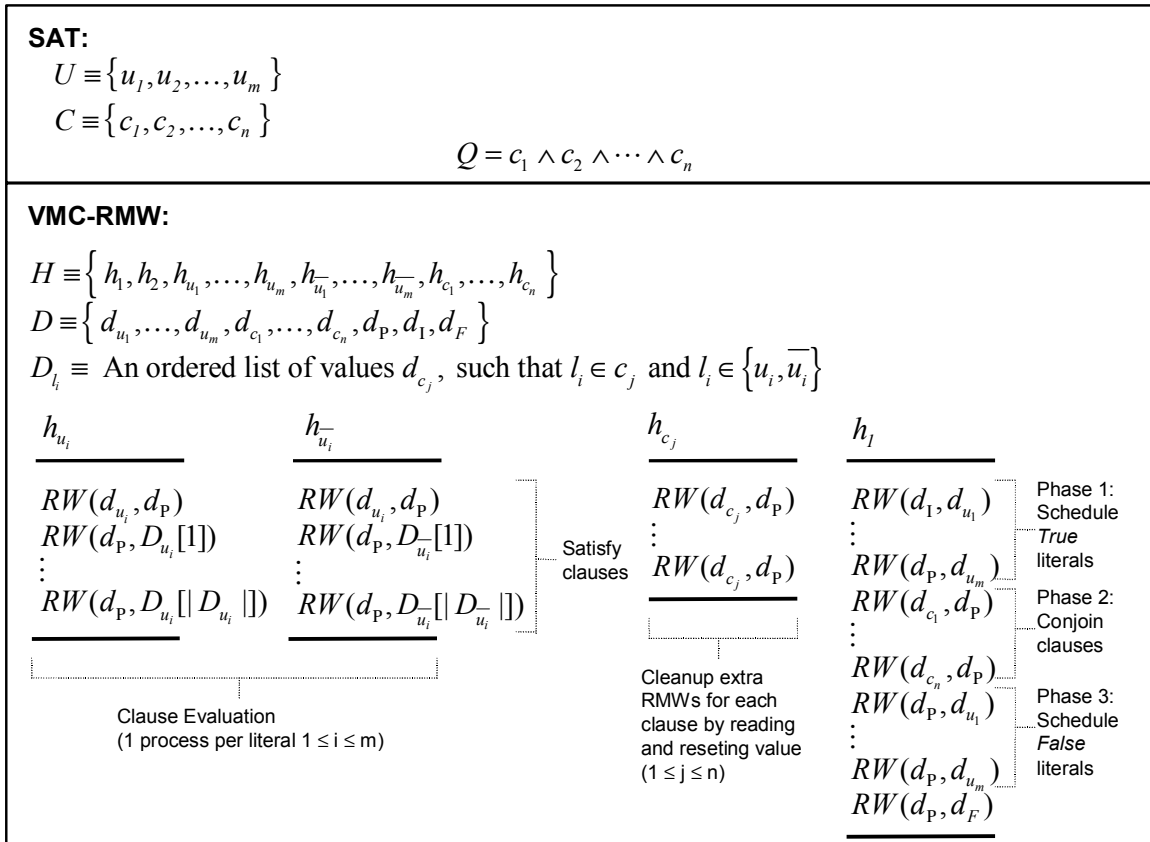


Figure 5.6: Reduction of SAT to VMC-RMW.

To better illustrate the transformation, we use the same example from Figures 5.2 and 5.3 in Figures 5.7 and 5.8 below. That is, an instance of SAT with two variables and two clauses, along with the corresponding instance of VMC-RMW.

SAT Instance:				
$U \equiv \{u_1, u_2\}$				
$C \equiv \{c_1 : \{u_1, u_2\}, c_2 : \{\bar{u}_1, u_2\}\}$				
$Q = (u_1 \vee u_2) \wedge (\bar{u}_1 \vee u_2)$				
VMC-RMW:				
$H \equiv \{h_1, h_{u_1}, h_{u_2}, h_{u_1}^-, h_{u_2}^-, h_{c_1}, h_{c_2}\}$				
$D \equiv \{d_{u_1}, d_{u_2}, d_{c_1}, d_{c_2}, d_p, d_I, d_F\}$				
$D_{u_1} \equiv \{d_{c_1}\} \quad D_{u_1}^- \equiv \{d_{c_2}\}$				
$D_{u_2} \equiv \{d_{c_1}, d_{c_2}\} \quad D_{u_2}^- \equiv \emptyset$				
$\frac{h_{u_1}}{\underline{\hspace{2cm}}}$	$\frac{h_{u_1}^-}{\underline{\hspace{2cm}}}$	$\frac{h_{c_1}}{\underline{\hspace{2cm}}}$	$\frac{h_{c_2}}{\underline{\hspace{2cm}}}$	$\frac{h_I}{\underline{\hspace{2cm}}}$
$\frac{RW(d_{u_1}, d_p)}{RW(d_p, d_{c_1})}$	$\frac{RW(d_{u_1}, d_p)}{RW(d_p, d_{c_2})}$	$\frac{RW(d_{c_1}, d_p)}{\underline{\hspace{2cm}}}$	$\frac{RW(d_{c_2}, d_p)}{\underline{\hspace{2cm}}}$	$\frac{RW(d_I, d_{u_1})}{RW(d_p, d_{u_2})}$
$\frac{RW(d_{u_2}, d_p)}{RW(d_p, d_{c_1})}$	$\frac{RW(d_{u_2}, d_p)}{RW(d_p, d_{c_2})}$			$\frac{RW(d_{c_1}, d_p)}{RW(d_{c_2}, d_p)}$
$\frac{RW(d_{u_2}, d_p)}{RW(d_p, d_{c_2})}$				$\frac{RW(d_p, d_{u_1})}{RW(d_p, d_{u_2})}$
				$\frac{RW(d_p, d_{u_2})}{RW(d_p, d_F)}$

Figure 5.7: Example SAT Instance and Corresponding Instance of VMC-RMW.

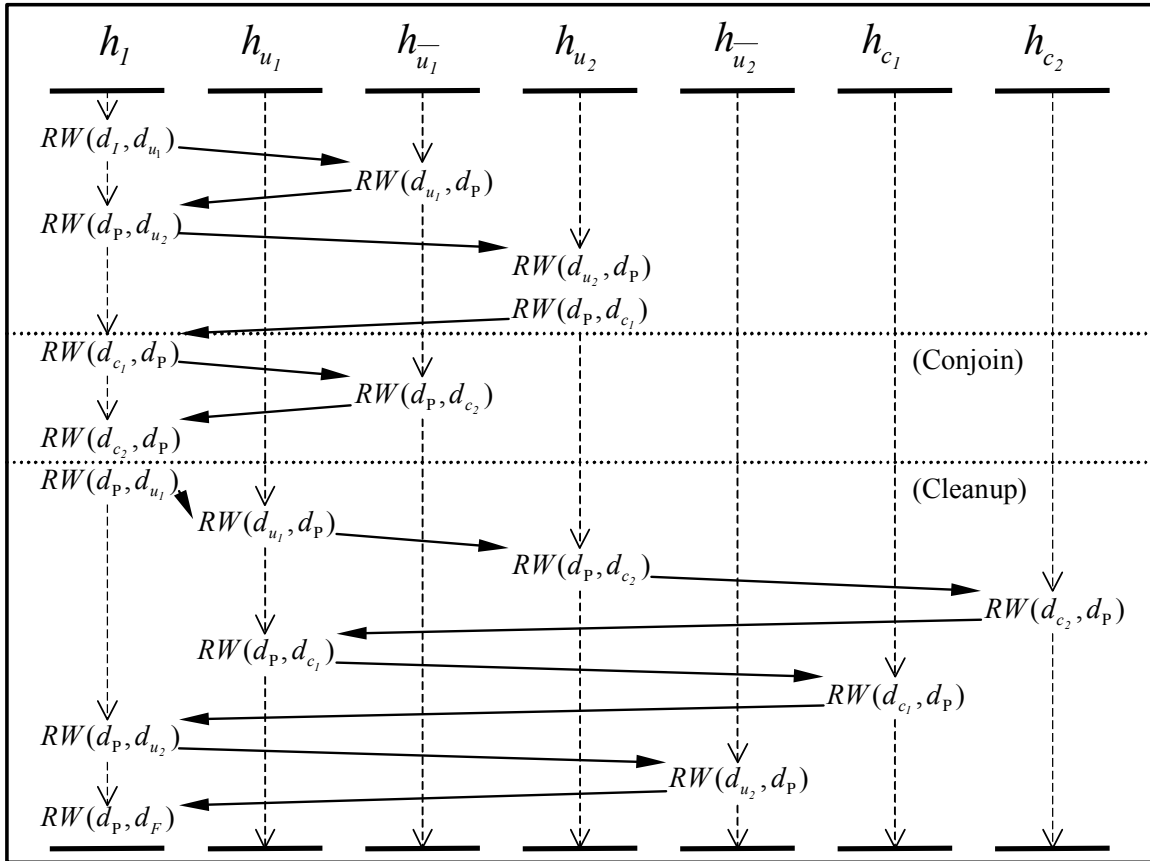


Figure 5.8: Previously Shown Instance of VMC-RMW with Coherent Schedule.

Here, the schedule corresponds to a truth assignment in which $u_1=false$ and $u_2=true$. The history h_l acts as a scheduler, enabling the first operation from one of each pair of literal histories to be scheduled. For these histories, additional operations may be scheduled in the conjoin phase (middle) that write values corresponding to the clauses. Under a satisfying assignment such as this one, a value may be written for every clause. These values are read by h_l , and then the cleanup phase begins (bottom).

THEOREM 5.5: VMC-RMW is NP-Complete

Proof:

1. Membership in NP:

A certificate for VMC-RMW is a schedule of all the memory operations. Given a schedule S for a set of process histories H , we can easily determine if it is a coher-

ent schedule. We can scan S to verify that it contains all the memory operations from H , in their respective process orders. While scanning S , we ensure that the read of each read-modify-write returns the value of the previous operation.

2. NP-Hardness:

Let Q be an arbitrary instance of SAT, and V the corresponding instance of VMC-RMW with set D of data values and set H of process histories as constructed above.

LEMMA 5.6: V is coherent if and only if Q is satisfiable.

- 1) Suppose V is coherent. By definition, there exists a coherent schedule S for H . For each variable u , the two process histories $\{h_u, h_{\bar{u}}\}$ must be interleaved in some way in S . This corresponds to a truth assignment T for U . Should the first operation from h_u precede the first operation $h_{\bar{u}}$ in S , the variable u is *true* under T , and *false* otherwise.

For each clause c in C , there is an operation $RW(d_c, d_p)$ in h_l that forces an operation $RW(*, d_c)$ to precede it in S . However, this only happens if H is such that an operation $RW(*, d_c)$ appears in a process history h_l for a literal l that is *true* under T . Note that we appended these operations to the literal histories in the same order the values are read in h_l . An operation $RW(*, d_c)$ appears in a process history h_l only if the clause c contains the literal l , and is thus satisfied by the truth of l . Hence, every clause in C is satisfied by T . Therefore, Q is satisfiable.

- 2) To prove the converse is true, suppose there is a satisfying truth assignment T for Q . Use T to interleave h_l with the first operation from each of the process histories' corresponding to literals over U .

By the construction, the process history for each literal over U contains an operation $RW(d_p, d_c)$ for each clause c in which it appears. Since T satisfies C , at least one literal per clause is *true* under T , and at least one of

the operations that write each value d_c may be scheduled before the operation that reads it in h_l .

Though a clause c may be satisfied by more than one literal, the history h_c can be interleaved with h_l to handle the “extra” operations that result. In addition, the history h_c allows the operations from process histories corresponding to literals that are false under T to be included in S . Therefore, it is possible to interleave all of H into a coherent schedule S , and Lemma 5.6 follows.

It is easy to see that the transformation can be implemented to run in polynomial-time reduction; hence VMC-RMW is NP-Complete. \square

5.3 General Case

We can now deduce the complexity of a more generalized statement of the problem. Let us allow combinations of simple operations, read-modify-writes, and other operations.

DEFINITION 5.7: Verifying Memory Coherence in General (General-VMC)

INSTANCE: Data value set D . Finite set H of process histories, each consisting of a finite sequence of reads, writes, and read-modify-writes with the same address.

QUESTION: Is there a coherent schedule S for H ?

COROLLARY 5.8: General-VMC is NP-Complete

Proof :

Included in the General-VMC are instances with only simple reads and writes of a single location. These instances form the VMC problem of Theorem 5.2. By the proof of Theorem 5.2, this problem is NP-Complete.

Since a subset of instances for the General-VMC problem form a known NP-Complete problem, and General-VMC is trivially in NP, it follows that General-VMC is NP-Complete. \square

Note that by the same technique we can include any of a number of other features. The NP-Completeness follows if simple reads, writes, or read-modify-writes are supported.

6 Complexity of Restricted and Augmented Cases

Once a problem is proven NP-Complete, an important task is to find the conditions under which it is tractable. We present a collection of results for restricted cases of verifying memory coherence, as well as the case in which the memory system has been augmented to provide the order of write operations.

For a number of cases, the complexity of VMC follows from previous work with sequential consistency [5,6]. For these cases, we refer to the previous work and only provide a reduction or algorithm if ours differs significantly.

6.1 Restricting Number of Memory Operations

We can restrict the number of memory operations in each process history to a small number. In other words, each process may only execute a small number of memory operations before the execution is paused and checked for coherence.

We find that the VMC problem remains NP-Complete for as few as three simple memory operations (reads and writes) per process. The problem is also NP-Complete with only two memory operations per process if all the operations are read-modify-writes [5]. However, if only one memory operation per process is allowed, the problem is in P [5]. The case for only two simple memory operations is left as an open problem.

First, let us examine the case where we have three simple memory operations per process. We can modify the reduction used for the general case as follows. Divide the histories h_1 and h_2 each into $\lceil m/3 \rceil$ histories with at most three write operations. Since the histories corresponding to literals have room for only one write operation after the two reads, we create a separate history for each clause the literal appears in (each with the same two reads as a prefix). The sequence of reads in h_3 is broken into several pieces, serialized by each writing a token value for the next to read. The cleanup phase of h_3 divides into m histories that each read the last token value, and performs two writes (rewriting the values written originally by h_1 and h_2). See Figure 6.1 below for the complete reduction.

$$H \equiv \left\{ h_{1,1}, \dots, h_{1, \lceil m/3 \rceil}, h_{2,1}, \dots, h_{2, \lceil m/3 \rceil}, h_{3,1}, \dots, h_{3, n+m}, h_{u_1,1}, \dots, h_{u_m, |D_{u_m}|}, h_{\bar{u}_1,1}, \dots, h_{\bar{u}_m, |D_{\bar{u}_m}|} \right\}$$

$$D \equiv \left\{ d_{u_1}, \dots, d_{u_m}, d_{\bar{u}_1}, \dots, d_{\bar{u}_m}, d_{c_1}, \dots, d_{c_n}, d_{t_1}, \dots, d_{t_n}, d_I, d_F \right\}$$

$$D_{l_i} \equiv \text{An ordered list of values } d_{c_j}, \text{ such that } l_i \in c_j \text{ and } l_i \in \{u_i, \bar{u}_i\}$$

$h_{1,1}$	$h_{2,1}$	$h_{u_i,1}$	$h_{\bar{u}_i,1}$	$h_{3,1}$	$h_{3, n+1}$
$W(d_{u_1})$	$W(d_{\bar{u}_1})$	$R(d_{u_i})$	$R(d_{\bar{u}_i})$	$R(d_{t_1})$	$R(d_{t_n})$
$W(d_{u_2})$	$W(d_{\bar{u}_2})$	$R(d_{\bar{u}_i})$	$R(d_{u_i})$	$R(d_{c_1})$	$W(d_{\bar{u}_1})$
$W(d_{u_3})$	$W(d_{\bar{u}_3})$	$W(D_{u_i}[1])$	$W(D_{\bar{u}_i}[1])$	$W(d_{t_1})$	$W(d_{\bar{u}_1})$
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
$h_{1, \lceil m/3 \rceil}$	$h_{2, \lceil m/3 \rceil}$	$h_{u_i, D_{u_i} }$	$h_{\bar{u}_i, D_{\bar{u}_i} }$	$h_{3, n}$	$h_{3, n+m}$
$W(d_{u_{m-2}})$	$W(d_{\bar{u}_{m-2}})$	$R(d_{u_i})$	$R(d_{\bar{u}_i})$	$R(d_{t_{n-1}})$	$R(d_{t_n})$
$W(d_{u_{m-1}})$	$W(d_{\bar{u}_{m-1}})$	$R(d_{\bar{u}_i})$	$R(d_{u_i})$	$R(d_{c_n})$	$W(d_{u_m})$
$W(d_{u_m})$	$W(d_{\bar{u}_m})$	$W(D_{u_i}[D_{u_i}])$	$W(D_{\bar{u}_i}[D_{\bar{u}_i}])$	$W(d_{t_n})$	$W(d_{\bar{u}_m})$
					$W(d_F)$

Figure 6.1: Reduction of SAT to VMC, Only Three Memory Operations Per Process.

THEOREM 6.1: VMC is NP-Complete with three memory operations per process

Proof (informal):

There are no dependences between the process histories representing h_1 and h_2 . They may be interleaved in any order to set the truth assignment. For the literals, there is a separate history for each clause the literal appears in, each with its own pair of reads such that they may all be scheduled if the literal is true under the assignment. The history h_3 is broken into pieces that must be scheduled consecutively due to the data dependence between them. As a result, all clauses must be satisfied to include them all. When this happens, the n^{th} (last) token d_{t_i} is written, and a set of process histories that perform cleanup writes may be scheduled. The cleanup process histories enable the histories for *false* literals to be scheduled, completing the sequence. \square

With as few as two operations per process history, the VMC-RMW problem remains NP-Complete. This result logically follows from an equivalent theorem in which VSC is

restricted to one address, and two read-modify-write operations per processor [5]. Since the reduction we found is different, it is illustrated in Figure 6.2 below.

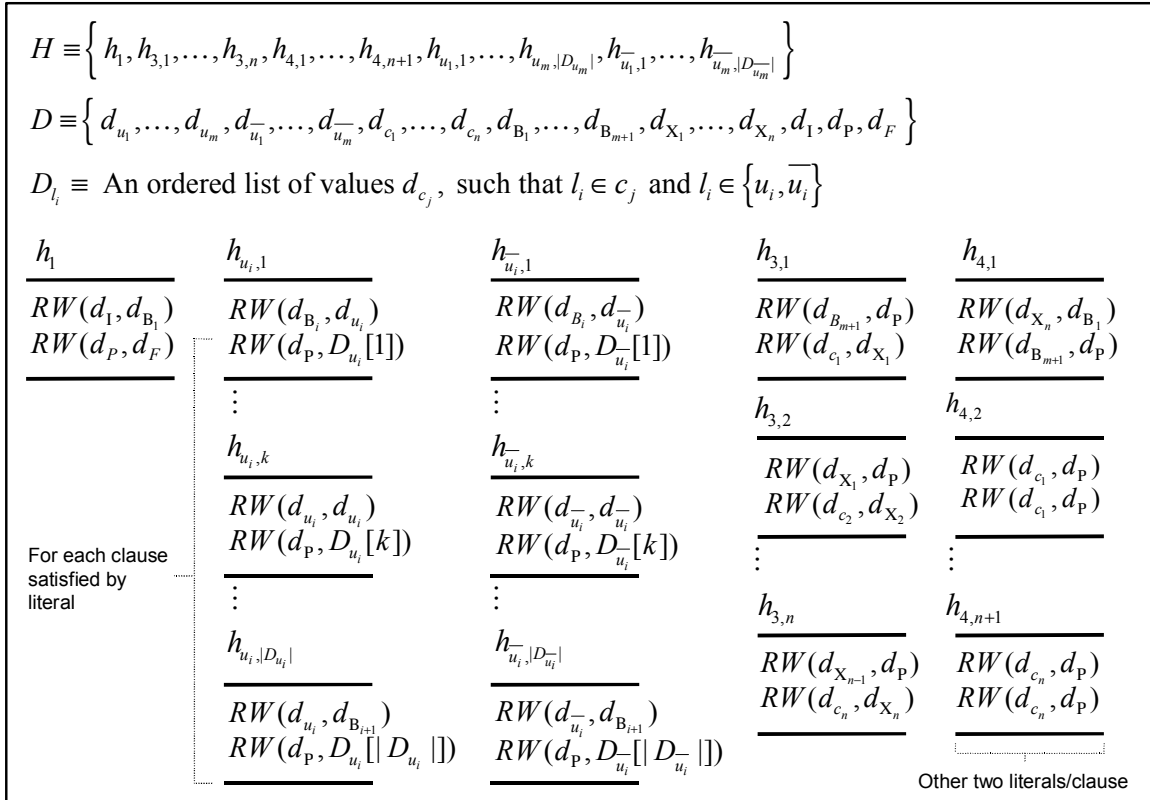


Figure 6.2: Reduction of 3SAT to VMC-RMW with Only Two Operations Per Process.

THEOREM 6.2: VMC-RMW is NP-Complete with two operations per process

Proof:

The VSC problem is NP-Complete when restricted to two read-modify-write operations per processor and one variable [5]. The VMC-RMW problem, when restricted to two memory operations is equivalent and therefore NP-Complete. \square

When restricted to only one memory operation per process, both the VMC and VMC-RMW problems are in P. This also follows from the fact that the VSC problem is in P when restricted to one memory operation per process [5]. Since that result holds for

multiple addresses, it follows that the subset of instances with only one address is no more difficult to solve.

THEOREM 6.3: VMC and VMC-RMW are in P when restricted to one memory operation per process history

Proof:

The VSC problem is in P when restricted to one memory operation per processor. Hence, there exists a polynomial-time algorithm for all instances with only one address. This includes all the instances of VMC with one memory operation per process history. A similar argument can be made for VMC-RMW. The theorem follows. \square

ALGORITHM 6.4: VMC with one operation per process history:

- 1 Separate the read and write operations of H into sets R and W , respectively.
- 2 Sort R and W such that the data values appear in the same order. The initial value should be first and the final value should be last in the sorted lists.
- 3 Remove all reads of the initial value from R , and place them at the beginning of S .
- 4 Remove the first write from W , and append it to S .
- 5 While the data of the first read in R matches the last operation in S match, remove the first read from R and append it to S .
- 6 While there are write operations in W , repeat steps 4-5.
- 7 If no reads remain in R , S is a coherent schedule for H .

The only way that a schedule may not exist is if there are read operations that return values not written by an operation in H . If this happens, there will be reads remaining in R when the algorithm terminates.

The first and third steps can be implemented to run in linear time. Together, steps 4 & 5 consume linear time, since they do something for each read and write. The sorting step can be implemented to run in $O(n \lg n)$ time. The algorithm has $O(n \lg n)$ complexity. For a memory system with m addresses, the bound becomes $O(m * n \lg n)$. \square

ALGORITHM 6.5: VMC-RMW with one operation per process history:

From H , we construct graph $G = \{V, E\}$ with vertex set V and set E of edges, as follows:

- 1 For each data value d_i read/written in H , add a vertex v_i to V .
- 2 For all read-modify-write operations $RW(d_i, d_j)$, add an edge $e = (v_i, v_j)$ to E .
- 3 If the initial value d_I does not match final value d_F , add an edge $e = (v_F, v_I)$ to E .

We now have a graph that is Eulerian if and only if there is a coherent schedule S for H . The edge from the final value to the initial value completes the cycle.

An Euler cycle with $n+1$ edges can be found in $O(|E|) = O(n^2)$ time. To convert the Euler cycle into a schedule S for H , we locate the edge $e = (v_F, v_I)$, break the cycle at that point, and traverse the path. Starting with the first edge leaving v_I , we replace each edge with a read-modify-write operation.

$$\forall i \in [1, n]: e_i = (d_j, d_k) \Rightarrow S[i] = RW(d_j, d_k) \quad (6.1)$$

The graph G can be constructed in $O(n^2)$ time for n read-modify-write operations. Once constructed, we can check for the existence of an Euler cycle in $O(|V|) = O(n)$ time by simply verifying that the number of incident edges equals the number of outgoing edges for every vertex. \square

6.2 Restricting Number of Processes

We can restrict the number of processes to a constant number. Real machines have limited scalability in terms of the number of processors, and similarly, the number of processes that participate in a parallel execution may be small.

This restriction was studied for sequential consistency in [6], and the complexity of VMC and VMC-RMW under this restriction follow from the results presented therein. The only difference is that coherence is a local property, which must be verified for each address individually.

THEOREM 6.6: VMC is in P when the number of processes is constant

Proof:

With n total memory operations, k processors and c addresses, the VSC problem can be solved in $O(n^k k^c)$ time [6]. The VMC problem with k processes is the subset of instances of the VSC problem in which $c=1$. Therefore, the complexity of the VMC problem is $O(n^k)$, which is polynomial for any constant k . To verify coherence for a system with c shared locations, the complexity is $O(cn^k)$. \square

THEOREM 6.7: VMC-RMW is in P when the number of processes is constant

Proof:

With n total memory operations, k processors, the VSC problem with only read-modify-write operations can be solved in $O(n^k)$ time [6]. The VMC-RMW problem with k processes is the subset of this problem with one address. Therefore, the complexity of the VMC-RMW problem is $O(n^k)$, which is polynomial for any constant k . Similar to the preceding proof, the verification cost is $O(cn^k)$, for a system with c shared locations. \square

6.3 Restricting Data Value Locality

We can also restrict the number of times a data value is written in an execution. Part of the complexity of verifying coherence comes from the fact that there may be many write operations that could have supplied data to a read operation.

We find that VMC becomes NP-Complete if data values are written at most twice, and VMC-RMW becomes NP-Complete if data values are written at most three times. If every value is written at most once, the problem is in P (analogous to adding a unique tag or timestamp to the data, or supplying the mapping of reads to writes [6]). The case for VMC-RMW with data values written at most twice remains an open problem.

THEOREM 6.8: VMC is in P when data values are written only once

Proof:

Restricted to one address, the VSC-Read problem is in P, with a linear time algorithm [6].

This is equivalent to the VMC problem when restricted such that data values may be written once, since for each read there can be only one write with the same data. It follows that the VMC problem is also in P, with a similar algorithm. \square

ALGORITHM 6.9: VMC-RMW with data values written only once

From H , construct graph $G=\{V, E\}$ with set V of vertices and set E of edges, as follows:

- 1 For each read-modify-write operation $RW_i(*,*)$ in H , add a vertex v_i to V .
- 2 For all pairs of read-modify-write operations that are adjacent in a process history, such that Op_i precedes Op_j add a program-order edge $e=(v_i, v_j)$ to E .
- 3 For all pairs of read-modify-write operations that are data-dependant, such that Op_i must precede Op_j , add a dependence edge $e=(v_i, v_j)$ to E .

We now have a graph with all of the necessary dependences for coherence. Since each value is written only once, each data dependence edge must be traversed in a path corresponding to a schedule. The program-order constraint is met if the graph is acyclic. The instance has a coherent schedule if, after removing the program-order edges, the corresponding graph G has a Hamiltonian path.

The graph can be constructed in $O(n^2)$ time, and checked for a cycle in $O(V+E)=O(n)$ time. We can traverse the graph to obtain a coherent schedule in $O(n)$ time. Therefore, when data values are written only once, VMC-RMW can be solved in $O(n^2)$ time. For a system with c shared locations, the complexity is $O(cn^2)$. \square

To prove that VMC is NP-Complete for instances with at most two writes of a value, we use a reduction modified from the proof of Theorem 5.2. For simplicity, we change the source problem to 3SAT to limit the number of literals per clause to three (See Appendix [23]). Then, for each clause c , we replace the unique data value d_c with a set of three unique values $\{d_{c,1}, d_{c,2}, d_{c,3}\}$, one for each of the three literals. The writes in each literal's history (for each satisfied clause) are updated to use the new values. We then replace the process history h_3 with a set of three histories. These enable a schedule to be constructed

if any one of the three values $\{d_{c,1}, d_{c,2}, d_{c,3}\}$ for each clause c is written before cleanup. Figure 6.3 shows the set of histories H used for the general 3SAT reduction.

$H \equiv \{ h_1, h_2, h_{3,1}, h_{3,2}, h_{3,3}, h_{u_1}, \dots, h_{u_m}, h_{u_1}^-, \dots, h_{u_m}^- \}$						
$D \equiv \{ d_{u_1}, \dots, d_{u_m}, d_{u_1}^-, \dots, d_{u_m}^-, d_{c_1}, \dots, d_{c_n}, d_I, d_F \}$						
$D_{l_i} \equiv$ An ordered list of values $d_{c_j,k}$ such that l_i is the k^{th} literal of c_j ($l_i \in \{u_i, \bar{u}_i\}$)						
h_1	h_2	h_{u_i}	$h_{u_i}^-$	$h_{3,1}$	$h_{3,2}$	$h_{3,3}$
$W(d_{u_1})$	$W(d_{u_1}^-)$	$R(d_{u_i})$	$R(d_{u_i}^-)$	$R(d_{c_{1,1}})$	$R(d_{c_{1,2}})$	$R(d_{c_{1,3}})$
$W(d_{u_2})$	$W(d_{u_2}^-)$	$R(d_{u_i}^-)$	$R(d_{u_i})$	$W(d_{c_{1,2}})$	$W(d_{c_{1,3}})$	$W(d_{c_{1,1}})$
\vdots	\vdots	$W(D_{u_i}[I])$	$W(D_{u_i}^-[I])$	\vdots	$R(d_{c_{2,2}})$	$R(d_{c_{2,3}})$
$W(d_{u_m})$	$W(d_{u_m}^-)$	\vdots	\vdots	$R(d_{c_{n,1}})$	$W(d_{c_{2,3}})$	$W(d_{c_{2,1}})$
		$W(D_{u_i}[\parallel D_{u_i} \parallel])$	$W(D_{u_i}^-[\parallel D_{u_i}^- \parallel])$	$W(d_{c_{n,2}})$	\vdots	\vdots
				$W(d_{u_1})$	$R(d_{c_{n,2}})$	$R(d_{c_{n,3}})$
				$W(d_{u_m})$	$W(d_{c_{n,3}})$	$W(d_{c_{n,1}})$
				$W(d_{u_1}^-)$		
				\vdots		
				$W(d_{u_m}^-)$		
				$W(d_F)$		

Figure 6.3: Reduction of 3SAT to VMC with Values Written At Most Twice.

THEOREM 6.10: VMC is NP-Complete when values are written at most twice.

Proof (informal):

The truth assignment and cleanup is the same as in the proof of Theorem 5.2. The only data values written more than twice were those of the subset of D that corresponds to the clauses in C . These have been replaced such that each clause c has three unique values $\{d_{c,1}, d_{c,2}, d_{c,3}\}$ associated with it, one for each literal. The history h_3 has been split into three histories such that the write of any one of these new values represents the satisfaction of the clause. Basically, one write enables a chain of operations to be scheduled that writes the other two values, always writing the data needed to schedule the corresponding read in $h_{3,1}$. If there is a satisfying truth assignment, the corresponding interleaving of h_1 and h_2 enables at least one write for every clause to be scheduled before cleanup,

enabling every read in $h_{3,l}$ to be scheduled. The cleanup phase takes care of all the remaining memory operations. \square

The technique developed in the proof of Theorem 6.10 above to conjoin the clauses does not work when all the operations are atomic read-modify-writes. To make a SAT reduction for the case of read-modify-writes (values each written at most three times), we modify the reduction used for Theorem 6.2. We add data values to serve the function of d_P in the original reduction, and exploit the fact that there is a separate history for each literal-clause combination to avoid writing some values a fourth time during cleanup. We then simplify the reduction, merging histories and removing unneeded values where possible. See Figure 6.4 for the complete reduction.

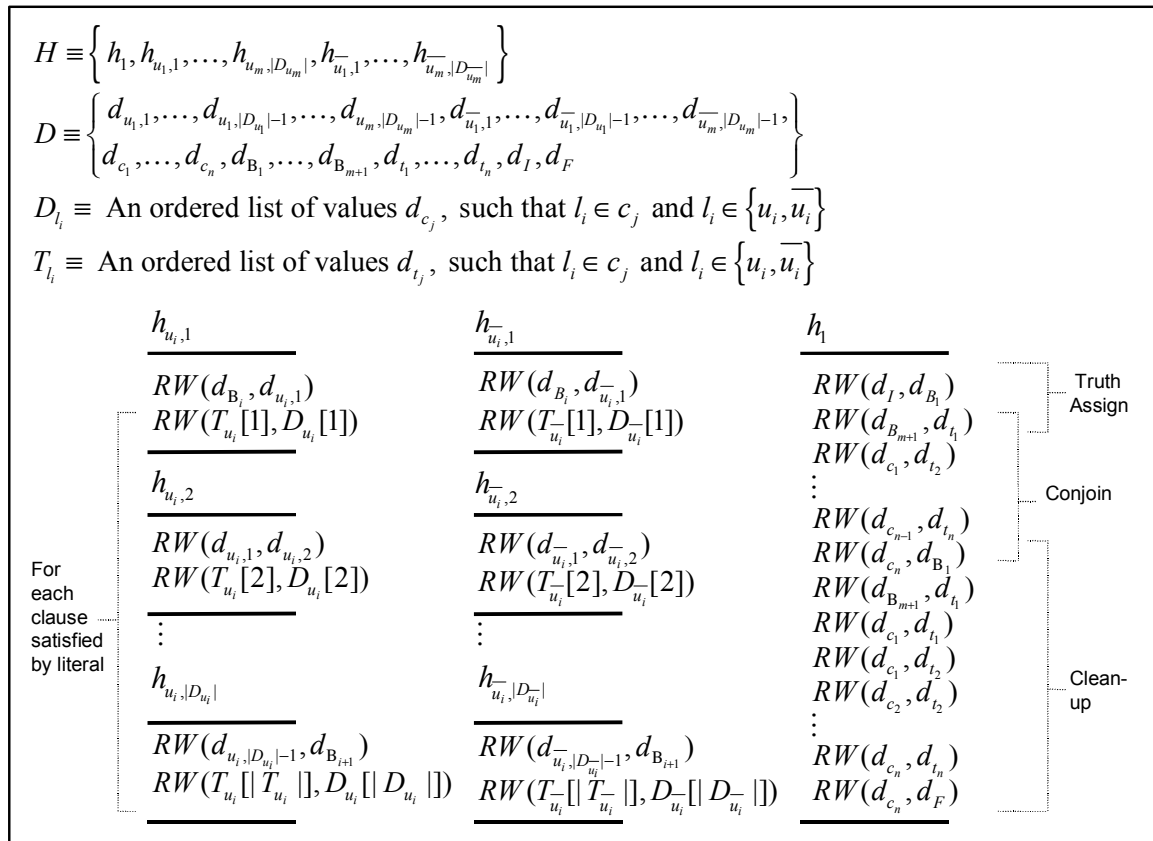


Figure 6.4: Reduction of 3SAT to VMC-RMW, Values Written at Most Three Times.

THEOREM 6.11: VMC-RMW is NP-Complete if values are written at most three times.

Proof (informal):

The process h_l first writes a token value d_B that enables operations to be scheduled that encode the truth assignment for the first variable. For each variable u , one of two sets of process histories (one for each literal) can be scheduled. Each history in the set has its first operation scheduled during the assignment (the second corresponds to a clause satisfied by the literal). The last history in a set writes the token value for the truth assignment of the next variable.

After all the variables have been assigned, h_l writes a different token value d_t to enable the scheduling of operations that account for all the clauses in C . For each clause c in C satisfied by a literal, there is a read-modify-write operation that reads a token d_t and writes d_c . The process h_l collects each of the special values d_c , writing the token for the next clause. Finally, cleanup is performed by rewriting the tokens used for the assignment, and collecting the two literals (per clause) not used earlier. There must be one literal assigned *true* in order to write the value d_c for each clause c . Therefore, we have a coherent schedule only under a satisfying assignment. \square

6.4 Supplying the Order of Writes

The memory system can be augmented to provide the order in which write operations were performed. In other words, if we can obtain a partial schedule from the ordering point in the system (or simulation), we can use it as a starting point for constructing a coherent schedule. Consistent with previous work, we will refer to such a partial schedule as the *write-order* [3,4,5,6].

We find that when the write-order is given, the VMC problem is in P with an algorithm that runs in $O(n^2)$ time for n simple operations. The case for read-modify-writes is trivially linear.

ALGORITHM 6.12: VMC with order of write operations given.

Assume we are given an instance H of VMC-Write with n memory operations, of which there are w write operations and r read operations ($n=r+w$). A sequence S_w of the w write operations in the order provided by the memory system ($S_w[0]$ has the initial value).

- 1 Create an empty array S of size $w*(r+1)+r$.
- 2 Insert each write $S_w[i]$, $1 \leq i \leq w$, into $S[(r+1)*i]$.
- 3 For each process history, set j equal to *zero* and do the following.
 - 4 For each read preceding the first write in a history, do:
 - 5 Compare the data to that of $S_w[j]$. If the data matches, insert the read into the first empty location after $S[0]$. Otherwise, increment j (exit if j indexes a write from the same process history).
 - 6 Starting with $k=1$, for each write $S_w[k]$, set j equal to k , and do:
 - 7 For each read after $S_w[k]$ in a process history, do:
 - 8 Compare the data to that of $S_w[j]$. If the data matches, insert the read into the first empty location after $S[(r+1)*j]$. Otherwise, increment j (exit if j indexes a write from the same process history as k).
- 9 To read out the completed sequence, initialize i to 0, read out operations from $S[i]$ until an empty location is found, then advance i to the value of $(\lfloor i/(r+1) \rfloor + 1)*(r+1)$.

Combined, the first two steps can be performed in $O(n)$ time. The third and fourth steps have $O(n^2)$ complexity (for each read, possibly checking each write, and then skipping over each read to find an insertion point). The last step can be performed in linear time. Overall, a schedule can be constructed in $O(n^2)$ time. \square

THEOREM 6.13: VMC-RMW with order of write operations given is in P.

Proof:

All read-modify-writes contain a write-component, such that the order of writes is a total

order on the operations. Thus, the write-order is the schedule. We can simply scan the schedule to check that program order is preserved, and that each read component returns the data of the last operations write component (in linear time). \square

6.5 Combining Restrictions

To gain further insight, we can combine the restrictions just investigated to form new variants of the problem. Any real implementation will have numerous restrictions that may simplify the problem.

Interestingly, when we restrict both the number of operations per process and the data value locality, we still have an NP-Complete problem. Specifically, VMC is NP-Complete for three memory operations per process and at most two writes of each data value, and VMC-RMW is NP-Complete for at most two memory operations per process and data values written at most three times. Hence, by combining restrictions we get a stronger result. See Figures 6.5 and 6.6 for the corresponding SAT reductions.

$$\begin{aligned}
H &\equiv \left\{ h_1, h_{2,1}, \dots, h_{2,n}, h_{3,1}, \dots, h_{3,n}, h_{u_1,1}, \dots, h_{u_m,|D_{u_m}|}, h_{u_1,1}^-, \dots, h_{u_m,|D_{u_m}|}^- \right\} \\
D &\equiv \left\{ d_{u_1,1}, \dots, d_{u_1,|D_{u_1}|-1}, \dots, d_{u_m,|D_{u_m}|-1}, d_{u_1,1}^-, \dots, d_{u_1,|D_{u_1}|-1}^-, \dots, d_{u_m,|D_{u_m}|-1}^- \right\} \\
&\quad \left\{ d_{c_1}, \dots, d_{c_n}, d_{B_1}, \dots, d_{B_{m+1}}, d_{t_1}, \dots, d_{t_n}, d_I, d_F \right\} \\
D_{l_i} &\equiv \text{An ordered list of values } d_{c_j} \text{ such that } l_i \in c_j \text{ and } l_i \in \{u_i, \bar{u}_i\} \\
T_{l_i} &\equiv \text{An ordered list of values } d_{t_j} \text{ such that } l_i \in c_j \text{ and } l_i \in \{u_i, \bar{u}_i\}
\end{aligned}$$

<u>$h_{u_i,1}$</u>	<u>$h_{u_i,1}^-$</u>	<u>h_1</u>	<u>$h_{3,1}$</u>
$RW(d_{B_i}, d_{u_i,1})$	$RW(d_{B_i}, d_{u_i,1}^-)$	$RW(d_I, d_{B_i})$	$RW(d_{c_1}, d_{t_1})$
<u>$RW(T_{u_i}[1], D_{u_i}[1])$</u>	<u>$RW(T_{u_i}^-[1], D_{u_i}^-[1])$</u>	<u>$RW(d_{B_{m+1}}, d_{t_1})$</u>	<u>$RW(d_{c_1}, d_{t_2})$</u>
<u>$h_{u_i,2}$</u>	<u>$h_{u_i,2}^-$</u>	<u>$h_{2,1}$</u>	\vdots
$RW(d_{u_i,1}, d_{u_i,2})$	$RW(d_{u_i,1}^-, d_{u_i,2}^-)$	<u>$RW(d_{c_1}, d_{t_2})$</u>	<u>$h_{3,n}$</u>
<u>$RW(T_{u_i}[2], D_{u_i}[2])$</u>	<u>$RW(T_{u_i}^-[2], D_{u_i}^-[2])$</u>	\vdots	$RW(d_{c_n}, d_{t_n})$
\vdots	\vdots	\vdots	<u>$RW(d_{c_n}, d_F)$</u>
<u>$h_{u_i, D_{u_i} }$</u>	<u>$h_{u_i, D_{u_i} }^-$</u>	<u>$h_{2,n}$</u>	
$RW(d_{u_i, D_{u_i} -1}, d_{B_{i+1}})$	$RW(d_{u_i, D_{u_i} -1}^-, d_{B_{i+1}}^-)$	$RW(d_{c_n}, d_{B_i})$	
<u>$RW(T_{u_i}[\lceil T_{u_i} \rceil], D_{u_i}[\lceil D_{u_i} \rceil])$</u>	<u>$RW(T_{u_i}^-[\lceil T_{u_i}^- \rceil], D_{u_i}^-[\lceil D_{u_i}^- \rceil])$</u>	<u>$RW(d_{B_{m+1}}, d_{t_1})$</u>	

Figure 6.5: 3SAT to VMC, Three Operations Per Process and Values Written Twice.

$$H \equiv \left\{ \begin{array}{l} h_{1,1}, \dots, h_{1, \lceil m/3 \rceil}, h_{2,1}, \dots, h_{2, \lceil m/3 \rceil}, h_{3,1,1}, \dots, h_{3,3,1}, \dots, h_{3,3,n}, \\ h_{4,1}, \dots, h_{4,m}, h_{u_1,1}, \dots, h_{u_m, |D_{u_m}|}, h_{u_1,1}, \dots, h_{u_m, |D_{u_m}|} \end{array} \right\}$$

$$D \equiv \left\{ d_{u_1}, \dots, d_{u_m}, d_{u_1}^-, \dots, d_{u_m}^-, d_{c_1,1}, \dots, d_{c_1,3}, \dots, d_{c_n,3} \right\}$$

$D_{l_i} \equiv$ An ordered list of values $d_{c_j, k}$ such that l_i is the k^{th} literal of c_j ($l_i \in \{u_i, \bar{u}_i\}$)

$h_{1,1}$	$h_{2,1}$	$h_{u_i,1}$	$h_{u_i,1}^-$	$h_{3,1,1}$	$h_{3,2,1}$	$h_{3,3,1}$	$h_{4,1}$
$W(d_{u_1})$	$W(d_{u_1}^-)$	$R(d_{u_i})$	$R(d_{u_i}^-)$	$R(d_{c_1,1})$	$R(d_{c_1,2})$	$R(d_{c_1,3})$	$R(d_{c_n,1})$
$W(d_{u_2})$	$W(d_{u_2}^-)$	$R(d_{u_i})$	$R(d_{u_i}^-)$	$W(d_{c_1,2})$	$W(d_{c_1,3})$	$W(d_{c_1,1})$	$W(d_{u_1})$
$W(d_{u_3})$	$W(d_{u_3}^-)$	$W(D_{u_i}[1])$	$W(D_{u_i}^-[1])$	\vdots	\vdots	\vdots	$W(d_{u_1}^-)$
\vdots	\vdots	\vdots	\vdots	$h_{3,1,n}$	$h_{3,2,n}$	$h_{3,3,n}$	\vdots
$h_{1, \lceil m/3 \rceil}$	$h_{2, \lceil m/3 \rceil}$	$h_{u_i, D_{u_i} }$	$h_{u_i, D_{u_i} }^-$	$R(d_{c_{n-1},1})$	$R(d_{c_n,2})$	$R(d_{c_n,3})$	$h_{4,m}$
$W(d_{u_{m-2}})$	$W(d_{u_{m-2}}^-)$	$R(d_{u_i})$	$R(d_{u_i}^-)$	$R(d_{c_{n-1},1})$	$W(d_{c_n,3})$	$W(d_{c_n,1})$	$R(d_{c_n,1})$
$W(d_{u_{m-1}})$	$W(d_{u_{m-1}}^-)$	$R(d_{u_i}^-)$	$R(d_{u_i}^-)$	$W(d_{c_n,2})$	---	---	$W(d_{u_m})$
$W(d_{u_m}^-)$	$W(d_{u_m}^-)$	$W(D_{u_i} [D_{u_i}])$	$W(D_{u_i}^- [D_{u_i}^-])$	---	---	---	$W(d_{u_m}^-)$

Figure 6.6: 3SAT to VMC-RMW, Two Operations and Values Written Three Times.

For all combinations of two or more restrictions in which one restriction yielded a polynomial time algorithm, the resulting case is trivially polynomial. For example, the case for a restricted number of processes ($O(n^k)$) and the write-order provided ($O(n^2)$) is trivially bounded to ($O(n^2)$) time. With some ingenuity, we may be able to obtain tighter bounds by exploiting the multiple restrictions. However, in the interest of space and time we leave the exhaustive examination of all possible combinations to future work.

6.6 Summary of Restricted Cases

The results presented in this section for each restricted case can be summarized in Figure 6.7 below (new results shaded).

Results for combinations of two or more restrictions are omitted so that a 2-dimensional table may be used. The case for two simple memory operations per process history remains an open problem. Note that the complexity of the case for read-modify-write operations where each value is written at most twice is also unknown.

	Simple Reads/Writes	Read-Modify-Writes
1 Operation/Process	$O(\text{nl}g(n))$	$O(n^2)$
2 Operations/Process	?	NP-Complete
3+ Operations/Process	NP-Complete	NP-Complete
Constant Processes	$O(n^k)$	$O(n^k)$
1 Write/Value	$O(n)$	$O(n^2)$
2 Writes/Value	NP-Complete	?
3+ Writes/Value	NP-Complete	NP-Complete
Write-order Given	$O(n^2)$	$O(n)$

Figure 6.7: Summary of Complexity Results for VMC.

Probably the most useful results are for the cases where the write-order is given, or the data values are made unique (via tags/timestamps). In these cases there are linear or $O(n^2)$ algorithms, and it may be possible to tighten these bounds further with other restrictions or clever algorithms. Thus, verifying memory coherence can be made tractable.

7 Beyond Coherence: Complexity of Verifying Consistency

In this section, we discuss how the NP-Completeness of verifying coherence implies the NP-Hardness of verifying adherence to memory consistency models that require coherence, including sequential consistency. We then extend our results to memory consistency models that do not strictly require coherence for all operations. Finally, we then show that verifying sequential consistency remains NP-Complete when executions are known to be coherent.

7.1 Sequential Consistency

As defined by Lamport, sequential consistency requires that there appear to be a serial order for all memory operation in which reads return the value written by the immediately preceding write with the same address, and the program orders of the different processes is respected [17]. This was stated formally in Section 4.2, and shown to be stronger than coherence.

To verify sequential consistency, we can find a consistent schedule for all the memory operations. Gibbons and Korach defined the task of finding such a schedule as the “Verifying Sequential Consistency” problem (VSC), and proved it is NP-Complete with a reduction from View-Serializability [4]. The same result can also be obtained from the complexity results of VMC, as shown below.

THEOREM 7.1: VSC is NP-Complete

Proof:

Restrict VSC to instances with only simple reads and writes of one memory location. This restricted version of VSC is equivalent to the VMC problem, which is NP-Complete by the proof of Theorem 5.2. Therefore, VSC is NP-Complete. \square

In other words, verifying sequential consistency is at least as hard as verifying memory coherence, since we can always compose programs that use only one shared variable. However, as one might expect, the multiplicity of addresses does add a degree of

difficulty to the problem of verifying consistency. This increases the complexity for some of the restricted cases. For example, VMC with a constant number of processes is in P, while the VSC problem is NP-Complete for as few as three processes [4]. Having the mapping of reads to writes reduces the VMC problem to polynomial complexity, though it is still NP-Complete for VSC [4].

7.2 Other Memory Consistency Models

There are many memory consistency models besides sequential consistency, including total store order (TSO), partial store order (PSO), processor consistency (PC), release consistency (RC), relaxed memory ordering (RMO), and weak ordering (WO). These weaker models enable hardware optimizations by relaxing ordering and atomicity constraints between memory operations.

All of the hardware-implemented memory consistency models, including those mentioned above, enforce a serial ordering on memory operations with the same address [14]. As with sequential consistency, we may compose programs that use only one shared variable in any such model. Each of these memory consistency models reduces to coherence for such cases, and is thus NP-Hard to verify. Verifying that executions provide consistency is therefore at least as difficult as verifying that they provide coherence.

There are memory consistency models that do not strictly require coherence, such as Lazy Release Consistency [24]. These models are typically used for shared virtual memory (SVM) systems, in which maintaining coherence for all locations and operations can incur a large communication overhead. In these systems, writes only need to be propagated and serialized if special synchronization instructions are used. Our results for memory coherence, however, can be extended to prove the NP-Hardness of verifying these models as well.

If memory operations to a given location are serialized by special instructions, we have the same basic problem as before. All memory operations must appear to execute in a serial order, respecting the data dependences between reads and writes. For Lazy

Release Consistency, we simply place an *acquire* operation and a *release* operation around each memory operation in the reductions of Section 5 (all with the same lock variable). As long as a consistency model provides a way to serialize memory operations, a reduction from SAT should be possible. This includes all currently known memory consistency models.

Now, it is important to point out that the NP-Hardness of verifying the known memory consistency models only applies to the general case. The precise conditions under which each model remains NP-Hard to verify has not been fully explored (only sequential consistency and linearizability have been thoroughly analyzed [3,4,5,6]). Some consistency models will be easier to verify adherence to than other models; knowledge of which may be of interest to designers of future systems. Much work remains to be done.

h_1	h_2	h_{u_i}	$h_{u_i}^-$	h_3
Acq	Acq	Acq	Acq	Acq
$W(d_{u_1})$	$W(d_{u_1}^-)$	$R(d_{u_1})$	$R(d_{u_1}^-)$	$R(d_{c_1})$
Rel	Rel	Rel	Rel	Rel
Acq	Acq	Acq	Acq	⋮
$W(d_{u_2})$	$W(d_{u_2}^-)$	$R(d_{u_1}^-)$	$R(d_{u_1})$	Acq
Rel	Rel	Rel	Rel	$R(d_{c_n})$
⋮	⋮	Acq	Acq	Rel
Acq	Acq	$W(D_{u_i}, [1])$	$W(D_{u_i}^-, [1])$	Acq
$W(d_{u_m})$	$W(d_{u_m}^-)$	Rel	Rel	$W(d_{u_i})$
Rel	Rel	⋮	⋮	Rel
⋮	⋮	Acq	Acq	⋮
⋮	⋮	$W(D_{u_i}, [D_{u_i}, I])$	$W(D_{u_i}^-, [D_{u_i}^-, I])$	Acq
⋮	⋮	Rel	Rel	$W(d_{u_m})$
⋮	⋮	⋮	⋮	Rel
⋮	⋮	⋮	⋮	Acq
⋮	⋮	⋮	⋮	$W(d_{u_i}^-)$
⋮	⋮	⋮	⋮	Rel
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	Acq
⋮	⋮	⋮	⋮	$W(d_{u_m})$
⋮	⋮	⋮	⋮	Rel
⋮	⋮	⋮	⋮	$W(d_F)$

Figure 7.1: SAT to VMC with Synchronization for Each Operation.

7.3 Complexity of Consistency with Coherence

The fact that all hardware-implemented memory consistency models provide coherence makes the verification of coherence both relevant and widely applicable. It suggests a two-step approach to dynamic verification of memory systems. First, coherence is verified; second, the additional requirements of the consistency model are checked.

This approach would be especially practical if verifying consistency was easier once coherence was verified. Therefore, we set out to determine if verifying consistency is tractable if it is known that the execution was coherent. We find that verifying sequential consistency is difficult in its own right, due to the multiplicity of shared variables to consider. With coherence removed from consideration, the VSC problem is still NP-Complete.

To reason about sequential consistency with coherence provided, we define a promise problem (VSCC). For a given instance, it is assumed that coherence has been provided, since there is no efficient way to check this restriction (ruling out a decision problem). Our complexity results only apply to such cases. All instances (positive and negative) are coherent.

DEFINITION 7.2: Verifying Sequential Consistency with Coherence (VSCC)

INSTANCE: Data value set D , address set A , finite set H of process histories, each consisting of a finite sequence of read and write operations.

PROMISE: For each address in A , there is a coherent schedule for H .

QUESTION: Is there a consistent schedule S for H ?

Assume we are given an arbitrary instance Q of SAT in conjunctive normal form, with set U of m variables and collection C of n clauses.

Create a set D of four unique data values. The first two values will be used for the truth assignment, while the third will be used to flag satisfied clauses. Assume all memory locations are initialized to d_l .

$$D \equiv \{ d_X, d_Y, d_Z, d_I \} \quad (7.1)$$

Next, create a set A of $m+n+I$ unique addresses, each for a distinct memory location. The first $m+n$ will represent the variables in the assignment and the clauses.

$$A \equiv \{ a_{u_1}, \dots, a_{u_m}, a_{c_1}, \dots, a_{c_n}, a_{\Delta} \} \quad (7.2)$$

Next, create two process histories, h_1 and h_2 . For each variable u , append a write operation $W(a_u, d_X)$ to h_1 and a write operation $W(a_u, d_Y)$ to h_2 . We will append more operations to h_1 and h_2 for cleanup in a later step.

$$\begin{aligned} h_1 &\equiv \langle W(a_{u_1}, d_X), W(a_{u_2}, d_X), \dots, W(a_{u_m}, d_X) \dots \rangle \\ h_2 &\equiv \langle W(a_{u_1}, d_Y), W(a_{u_2}, d_Y), \dots, W(a_{u_m}, d_Y) \dots \rangle \end{aligned} \quad (7.3)$$

Note that with no data dependences between h_1 and h_2 , all 2^m partial orders between each pair of writes are possible in a sequence. These partial orders will encode the truth assignment (T) for U :

$$\begin{aligned} T : U &\mapsto \{True, False\} \\ W(a_{u_i}, d_X) &\xrightarrow{\text{dyn}} W(a_{u_i}, d_Y) \Leftrightarrow T(u_i) = True \\ W(a_{u_i}, d_Y) &\xrightarrow{\text{dyn}} W(a_{u_i}, d_X) \Leftrightarrow T(\bar{u}_i) = True \end{aligned} \quad (7.4)$$

For each literal over U , define a subset of A with elements a_c representing each clause c that includes the literal.

$$\begin{aligned}
A_{u_i} &\equiv \text{An ordered list of values } a_{c_j} \text{ such that } u_i \in c_j \\
A_{\bar{u}_i} &\equiv \text{An ordered list of values } a_{c_j} \text{ such that } \bar{u}_i \in c_j
\end{aligned} \tag{7.5}$$

For each literal u , create a process history h_u , with a read operation $R(a_u, d_X)$ followed by another read operation $R(a_u, d_Y)$. After the read operations, add write operations of the value d_Z with addresses corresponding to each clause in which the literal appears (in order starting with the first). Construct a similar history, $h_{\bar{u}}$, for the complementary literal, only with the read operations in the reverse order.

$$\begin{aligned}
&\forall u_i \in U : \\
h_{u_i} &\equiv \langle R(a_{u_i}, d_X), R(a_{u_i}, d_Y), W(A_{u_i}[1], d_Z), \dots, W(A_{u_i}[|A_{u_i}|], d_Z) \rangle \\
h_{\bar{u}_i} &\equiv \langle R(a_{u_i}, d_Y), R(a_{u_i}, d_X), W(A_{\bar{u}_i}[1], d_Z), \dots, W(A_{\bar{u}_i}[|A_{\bar{u}_i}|], d_Z) \rangle
\end{aligned} \tag{7.6}$$

Next, create a process history h_3 , with read operations that can be scheduled only if all the clauses have been satisfied. For each c in C , append a read operation $R(a_c, d_Z)$ to h_3 . To indicate a satisfying instance, end the history with a write to a special location a_Δ .

$$h_3 \equiv \langle R(a_{c_1}, d_Z), R(a_{c_2}, d_Z), \dots, R(a_{c_n}, d_Z), W(a_\Delta, d_Z) \rangle \tag{7.7}$$

Since only half of the literals are *true* for any truth assignment of U , we need to handle the remaining memory operations once the read operations in h_3 have been scheduled (a “cleanup” phase). For reasons that will become clearer later, we add extra writes to h_1 and h_2 , unlike previous constructions where they were put in h_3 .

For cleanup, we first add a read of a_Δ to ensure that all the reads in h_3 are scheduled before the extra writes. We then have h_1 and h_2 write, to each variable’s location, the opposite value written previously.

$$\begin{aligned}
h_1 &\equiv \left\langle \begin{array}{l} W(a_{u_1}, d_X), \dots, W(a_{u_m}, d_X), R(a_\Delta, d_Z), \\ W(a_{u_1}, d_Y), \dots, W(a_{u_m}, d_Y) \end{array} \right\rangle \\
h_2 &\equiv \left\langle \begin{array}{l} W(a_{u_1}, d_Y), \dots, W(a_{u_m}, d_Y), R(a_\Delta, d_Z), \\ W(a_{u_1}, d_X), \dots, W(a_{u_m}, d_X) \end{array} \right\rangle
\end{aligned} \tag{7.8}$$

The reduction is now complete; let H be the set of process histories just constructed. See Figure 7.2 for a summary of the reduction.

$$H \equiv \{ h_1, h_2, h_3, h_{u_1}, \dots, h_{u_m}, h_{u_1}^-, \dots, h_{u_m}^- \} \tag{7.9}$$

SAT: $U \equiv \{ u_1, u_2, \dots, u_m \}$ $C \equiv \{ c_1, c_2, \dots, c_n \}$ $Q = c_1 \wedge c_2 \wedge \dots \wedge c_n$				
VSCC: $H \equiv \{ h_1, h_2, h_3, h_{u_1}, \dots, h_{u_m}, h_{u_1}^-, \dots, h_{u_m}^- \}$ $D \equiv \{ d_X, d_Y, d_Z, d_I \}$ $A \equiv \{ a_{u_1}, \dots, a_{u_m}, a_{c_1}, \dots, a_{c_n}, a_\Delta \}$ $A_{l_i} \equiv$ An ordered list of values a_{c_j} such that $l_i \in c_j$ and $l_i \in \{ u_i, \bar{u}_i \}$				
h_1	h_2	h_{u_i}	$h_{u_i}^-$	h_3
$W(a_{u_1}, d_X)$	$W(a_{u_1}, d_Y)$	$R(a_{u_i}, d_X)$	$R(a_{u_i}, d_Y)$	$R(a_{c_1}, d_Z)$
\vdots	\vdots	$R(a_{u_i}, d_Y)$	$R(a_{u_i}, d_X)$	$R(a_{c_2}, d_Z)$
$W(a_{u_m}, d_X)$	$W(a_{u_m}, d_Y)$	$W(A_{u_i}[1], d_Z)$	$W(A_{u_i}^-[1], d_Z)$	\vdots
$R(a_\Delta, d_Z)$	$R(a_\Delta, d_Z)$	\vdots	\vdots	$R(a_{c_n}, d_Z)$
$W(a_{u_1}, d_Y)$	$W(a_{u_1}, d_X)$	$W(A_{u_i}[[A_{u_i}]], d_Z)$	$W(A_{u_i}^-[[A_{u_i}^-]], d_Z)$	$W(a_\Delta, d_Z)$
\vdots	\vdots	\vdots	\vdots	\vdots
$W(a_{u_m}, d_Y)$	$W(a_{u_m}, d_X)$	\vdots	\vdots	\vdots

Figure 7.2: Reduction of SAT to VSCC.

It is easy to see that this transformation can be implemented to run in polynomial time. With m variables and n clauses in Q , the resulting VSCC instance has $2m+3$ process histories and $O(mn)$ memory operations.

THEOREM 7.3: VSCC is NP-Complete

Proof:

1. Membership in NP:

A certificate for VSCC is a schedule of all the memory operations. As with the VSC problem defined in [4], given such a schedule we can check that it is consistent in polynomial time.

2. NP-Hardness:

Let Q be an arbitrary instance of SAT, and V the corresponding instance of VSCC with set D of data values and set H of process histories constructed above.

LEMMA 7.4: V is sequentially consistent if and only if Q is satisfiable.

Proof:

1) Suppose V is sequentially consistent. By definition there is a consistent schedule S for H . For each variable u , the first two corresponding write operations $\{W(a_u, d_x), W(a_u, d_y)\}$ from h_1 and h_2 must appear in S in some order. By equation (7.4), the order encodes the truth assignment T for U . If u is assigned *true*, we can schedule the process history h_u , and the first read operation of the complementary literal's history, $h_{\bar{u}}$. Conversely, if u is assigned *false*, we can schedule the process history $h_{\bar{u}}$, and the first read operation of h_u . In either case, to completely schedule the history for the other (*false*) literal's history, the value returned by the second read must be rewritten. However this will not happen until after the write to address a_{Δ} , in the cleanup phase.

The write operations that correspond to the clauses in C are in the suffixes of the histories corresponding to the literals, such that a write op-

eration $W(a_c, d_z)$ can only appear before the cleanup phase in S if a literal in clause c is *true* under T . Because a clause is a disjunction of literals, this means the clause c is satisfied under T .

The data dependences created by the read operations of h_3 require a write operation $W(a_c, d_z)$ for each clause in C to appear before the cleanup phase. Note that we appended the write operations to the literal histories in the same order as the values are read. Hence, every clause in C must be satisfied by T . Therefore, Q is satisfiable.

- 2) To prove the converse is true, suppose Q is satisfiable. There is a satisfying truth assignment T for C . Use T to interleave h_1 and h_2 as defined in equation (7.4). The set of literals assigned *true* correspond exactly to the set of process histories that may be interleaved with the writes from h_1 and h_2 that precede $R(a_\Delta, d_z)$.

Since T satisfies C , at least one literal per clause is *true* under T . By the construction, at least one of the process histories containing each $W(a_c, d_z)$ for each clause in C may be scheduled before the cleanup phase in S . Hence, all the read operations $R(a_c, d_z)$ in h_3 may be included in S once all of the histories corresponding to literals that are *true* under T have been scheduled.

With all the read operations in h_3 scheduled, we are free to schedule the write $W(a_\Delta, d_z)$ that marks the beginning of the cleanup phase. The second set of writes in h_1 and h_2 may be interleaved however necessary to provide data for the read operations in the remaining (*false*) literals' process histories. Therefore, we have a sequentially consistent schedule S for H, V is sequentially consistent, and Lemma 7.4 follows.

Since VSCC is in NP, and SAT reduces to VSCC in polynomial time, it follows that VSCC is NP-Complete. \square

To verify coherence for this construction, separate the memory operations in H by address, as shown in Figure 7.3. Based on the particular address, we have one of three cases. In the first case, we have the memory operations used to assign and test the truth of a variable, for which a schedule is always possible (interleave the uncomplemented literal's history with h_1 , interleave the complemented literal's history with h_2 , and append one of the resulting schedules to the other). In the other two cases, only one value is read from or written to the location (and written at least once), so we may trivially schedule the read(s) after the write(s).

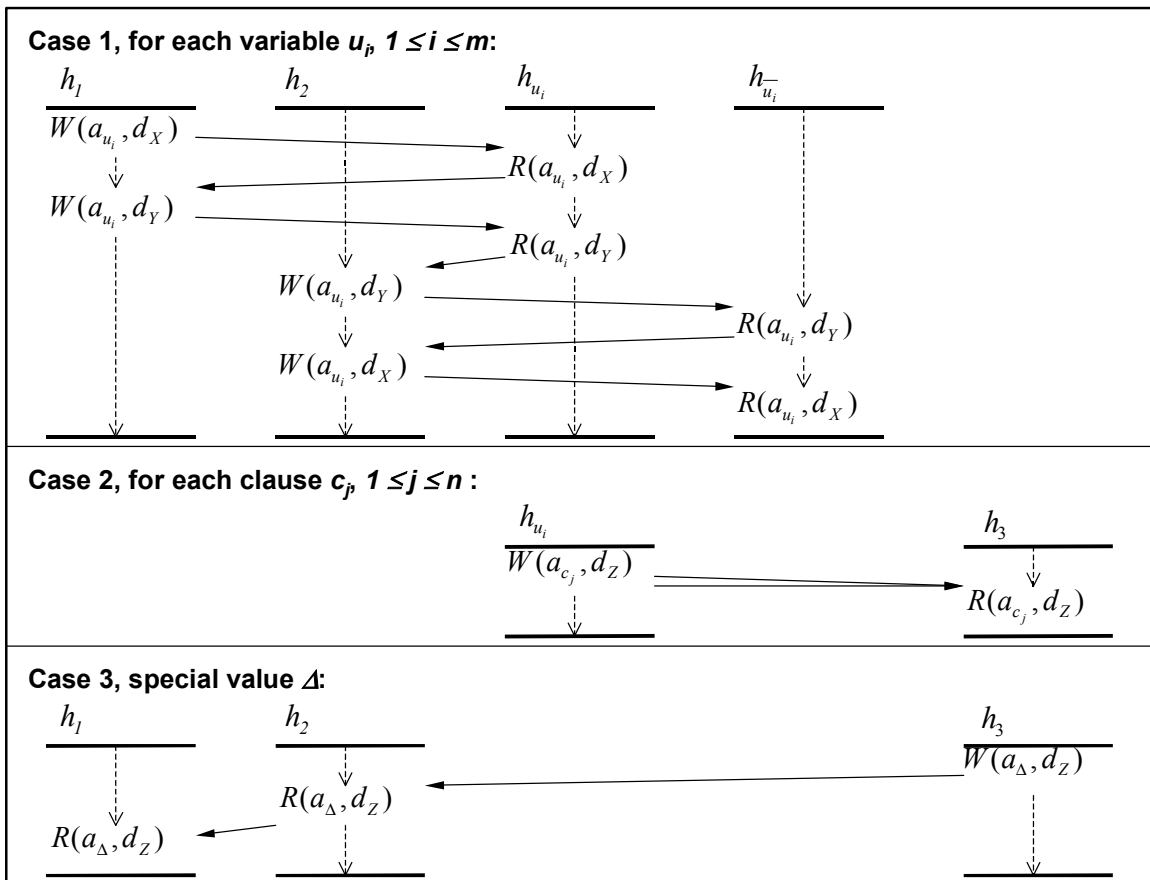


Figure 7.3: VSCC with Memory Operations Separated by Address.

In all three cases, there is a coherent schedule independent of the satisfiability of the original SAT instance. Further, it is easy to see that we can construct a coherent schedule for each address in polynomial time. Yet, by the proof of Theorem 7.2, the VSCC

problem is NP-Complete. Hence, knowing that coherence was provided for the execution does not reduce the complexity of verifying sequential consistency. This may be true for other consistency models as well.

Furthermore, we can constrain the problem such that coherence may be efficiently verified. Recall that in Section 6.4, we examined the complexity of VMC when the order in which write operations were executed is provided, and found it to be tractable. With the write-order, the VSCC problem is reduced to a decision problem in which we can first verify coherence. However, it was proven by Gibbons and Korach that the VSC problem remains NP-Complete when the write-order is provided for each location [3,4,5,6]. Thus, even the ability to efficiently verify coherence does not imply that verifying sequential consistency is tractable.

Going still further, we can use the schedules constructed while verifying memory coherence as an input to the VSC problem. Embedded in a coherent schedule is an order in which write operations were executed for each location, and a mapping from read operations to write operations with the same data value. It was shown previously that this information can be used to generate a sequentially consistent schedule in $O(nlgn)$ time (the VSC-Conflict problem) [3,4,5,6]. The catch is that this is achieved by treating the coherent schedules as a constraint. There may be many different sets of coherent schedules for an execution, yet only a few that can be combined into a sequentially consistent schedule. Hence, the failure to find a sequentially consistent schedule may only mean that the wrong set of coherent schedules were produced when verifying coherence. Like other NP-Complete problems, VSC is resistant to divide-and-conquer approaches.

8 Closing The Loop

Despite the NP-Completeness of VMC, it is still important to have algorithms for the general case. Depending on the application, such algorithms may perform acceptably for the size and structure of the instances involved. Note that the instances used in the SAT reductions of previous sections were contrived, and unlikely to occur in a real implementation or simulation.

In this section, we show two techniques for finding a coherent schedule. First, we present a simple recursive algorithm, with exponential run time in the worst case. Second, we discuss some simple modifications and heuristics that may improve the algorithm's run time for common cases. Finally, we present a transformation from VMC to SAT, enabling us to exploit the many optimized algorithms that already exist for SAT [25,26]. However, the transformation yields SAT instances that are relatively large compared to the corresponding VMC instances.

8.1 Simple Algorithm

Here, we describe a simple recursive algorithm that finds a coherent schedule if one exists. The algorithm may try all possible schedules, consuming exponential time in the worst case. However, such behavior is probably uncommon for process histories from real executions or simulations.

ALGORITHM 8.1: VMC

Find_Schedule(H, d_I, d_F):

- 1 Initialize sequence S to empty
- 2 $Success \leftarrow$ Recursive_Interleave(H, S, d_I, d_F)
- 3 If $Success=$ True, then return S , else return empty sequence.

Recursive_Interleave(H, S, d, d_F):

- 1 For each history h in H , do the following:

```

2       While next unscheduled operation is a Read of value  $d$ , do:
3           Append next unscheduled operation to  $S$ .
4       If all memory operations have been scheduled, and  $d = d_F$ , then return (True).
5       For each history  $h$  in  $H$ , do the following:
6           If next unscheduled operation in  $h$  is a Write, then:
7               Add next unscheduled operation to  $S$ , set  $d$  to data written
8                $Success \leftarrow \text{Recursive\_Interleave}(H, S, d, d_F)$ 
9               If  $Success = \text{True}$ , then return (True).
10          Remove operations just appended to  $S$ 
11     Return (False)

```

This algorithm starts with the first operation in each history, and tries to schedule as many read operations as possible using the initial value. It then (arbitrarily) chooses a write operation, and calls itself. Each recursive call attempts to schedule as many read operations with the data from the last write operation, and then arbitrarily selects the next write operation. If a point is reached where all operations are scheduled, and the final value matches the data of the last write, a schedule has been found. If not, or if it is not possible to schedule the next operation (all reads are of data other than that of last write), the algorithm backtracks to the last write, and selects a different one. The algorithm may backtrack repeatedly until a level of recursion is reached in which there exists a write that has not yet been tried as the next write in a potential schedule. If the first write in each history has been tried (unsuccessfully) as the first write in a schedule, a schedule does not exist.

8.2 Optimizing the Simple Algorithm

Though simple, the algorithm shown above leaves much room for improvement. By choosing the next write for the schedule arbitrarily, the algorithm may make poor choices. Poor choices that are made early in the construction of a schedule are very costly, since the algorithm will try all possibilities for succeeding writes before conclud-

ing that a bad choice was made. However, there are some simple heuristics to help the algorithm make better decisions.

First, the algorithm can examine all the not-yet-scheduled read operations from the other process histories before choosing the next write operation. A greedy heuristic is to choose the write operation that will enable the most read operations to be scheduled later. Programs typically perform more reads than writes, and values are often read before they are written. There may not be many writes to choose from at any given step, and it is possible that the data of the next write operation was returned by read operations in other processes.

Second, we can augment the algorithm to look ahead in the process histories some bounded number of operations. For example, by looking ahead one operation, the algorithm can ensure that the data of the next write operation is not returned after that of some other unscheduled write. Looking ahead more than one operation, the algorithm can try to construct a partial schedule that contains as many unscheduled reads as possible. Combined with the first heuristic, the algorithm might choose to schedule the write that enables a maximum number of reads to be scheduled, and is unlikely to be preceded by another unscheduled write.

Finally, we can do some preprocessing on the process histories to remove unnecessary complexity from the problem. First, pairs of consecutive reads returning the same data can be merged into one operation. If we scan the process histories and construct sets for the data values read/written, we can identify and remove some dead writes from the process histories.

8.3 Reduction to SAT

There has been much work developing algorithms for SAT problems [25,26]. The state of the art algorithms can tackle instances with thousands of variables [25]. By converting VMC to SAT, we can make use of these optimized algorithms.

Given an instance H of VMC with n total memory operations ($n > 1$), we can construct a SAT instance $Q = \{U, C\}$, such that there exists a satisfying truth assignment if and only if there is a coherent schedule S for H .

First, allocate set U of n^2 Boolean variables. Let the truth of a variable $U[i][j]$ indicate that the i^{th} position in the schedule is occupied by the j^{th} memory operation ($i, j \in [1, n]$).

Next, define a set of clauses C to ensure each of the following constraints is met: 1) Each memory operation is assigned at least one position in the schedule. 2) Each memory operation is assigned at most one position in the schedule. 3) Memory operations from the same process history appear in program order. 4) Read operations are immediately preceded by memory operations that read or write the same data value. 5) The last memory operation has the final data value.

For simplicity, we will separately define a subset of clauses for each of the five constraints outlined above, and C will be the union of these subsets.

$$C \equiv C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5 \quad (8.1)$$

Subset C_1 has n clauses with n literals per clause, one for each memory operation, that are satisfied if the memory operations each occupy at least one position in the schedule.

$$C_1 \equiv \{U[1][i] \vee U[2][i] \vee \dots \vee U[n][i] : i \in [1, n]\} \quad (8.2)$$

Next, subset C_2 has $(n^3 + n^2 + 2n)/2 = O(n^3)$ clauses with two literals each, all satisfied if no more than one memory operation has been assigned to each position in the schedule.

$$C_2 \equiv \bigcup_{i=1}^n \left\{ \begin{array}{l} (\overline{U[i][1]} \vee \overline{U[i][2]}), \dots, (\overline{U[i][1]} \vee \overline{U[i][n]}), \dots \\ (\overline{U[i][j]} \vee \overline{U[i][j+1]}), \dots, (\overline{U[i][j]} \vee \overline{U[i][n]}), \dots \\ (\overline{U[i][n-1]} \vee \overline{U[i][n]}) \end{array} \right\} \quad (8.3)$$

Subset C_3 has $n(n-1)=O(n^2)$ clauses with at most $n-1$ literals in each. These clauses are satisfied if, for each memory operation, the immediately preceding memory operation from the same process precedes it in the schedule (occupies a position with a lower index). This expresses the program order constraint.

$$C_3 \equiv \left\{ \begin{array}{l} \left(\overline{U[i][j]} \vee U[i-1][k] \vee \dots \vee U[1][k] \right) : \\ i \in (1, n) \wedge (Op_k \xrightarrow{po} Op_j) \wedge \neg(\exists l : Op_k \xrightarrow{po} Op_l \wedge Op_l \xrightarrow{po} Op_j) \end{array} \right\} \quad (8.4)$$

Subset C_4 has at most n clauses, one for each read operation, that ensure an operation with the same data value immediately precedes it in the schedule (next lower index). To handle the boundary case of reads that return the initial value, create a write operation with the initial value, and assign it to the 0^{th} position in the schedule.

$$M_d \equiv \{j : (Op_j = R(d) \vee Op_j = W(d))\}$$

$$C_4 \equiv \left\{ \begin{array}{l} \left(\overline{U[k][i]} \vee U[k-1][M_d[1]] \vee \dots \vee U[k-1][M_d[|M_d|]] \right) : \\ k \in [1, n] \wedge (Op_i = R(d)) \end{array} \right\} \quad (8.5)$$

Finally, subset C_5 has one clause, with at most n literals. We take the set of memory operations that read/write the final data value, and require that one of them occupy the last position in the schedule.

$$M_F \equiv \{i : Op_i = R(d_F) \vee Op_i = W(d_F)\} \quad (8.6)$$

$$C_5 \equiv \{U[n][M_F[1]] \vee U[n][M_F[2]] \vee \dots \vee U[n][M_F[|M_F|]]\}$$

This completes the construction of the SAT instance. For an instance H of VMC with n total memory operations, the corresponding instance Q of SAT has n^2 variables and $O(n^3)$

clauses (some with $O(n)$ literals). Figure 8.1 shows an example of a simple VMC instance and the corresponding SAT instance.

VMC:	h_1	h_2	h_3
$H \equiv \{h_1, h_2, h_3\}$	<u> </u>	<u> </u>	<u> </u>
$D \equiv \{d_1, d_2, d_I\}$	<u> </u>	<u> </u>	<u> </u>
$d_F = d_2$	<u> </u>	<u> </u>	<u> </u>
SAT:			
$U \equiv \left\{ \begin{array}{l} u_{W(d_1),1}, u_{W(d_1),2}, u_{W(d_1),3}, u_{W(d_1),4}, u_{W(d_2),1}, u_{W(d_2),2}, u_{W(d_2),3}, u_{W(d_2),4}, \\ u_{R(d_1),1}, u_{R(d_1),2}, u_{R(d_1),3}, u_{R(d_1),4}, u_{R(d_2),1}, u_{R(d_2),2}, u_{R(d_2),3}, u_{R(d_2),4} \end{array} \right\}$			
$C \equiv C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5$			
$C_1 \equiv \left\{ \begin{array}{l} (u_{W(d_1),1} \vee u_{W(d_1),2} \vee u_{W(d_1),3} \vee u_{W(d_1),4}), (u_{W(d_2),1} \vee u_{W(d_2),2} \vee u_{W(d_2),3} \vee u_{W(d_2),4}), \\ (u_{R(d_1),1} \vee u_{R(d_1),2} \vee u_{R(d_1),3} \vee u_{R(d_1),4}), (u_{R(d_2),1} \vee u_{R(d_2),2} \vee u_{R(d_2),3} \vee u_{R(d_2),4}) \end{array} \right\}$			
$C_2 \equiv \left\{ \begin{array}{l} (\overline{u_{W(d_1),1}} \vee \overline{u_{W(d_2),1}}), (\overline{u_{W(d_1),1}} \vee \overline{u_{R(d_1),1}}), (\overline{u_{W(d_1),1}} \vee \overline{u_{R(d_2),1}}), (\overline{u_{W(d_2),1}} \vee \overline{u_{R(d_1),1}}), (\overline{u_{W(d_2),1}} \vee \overline{u_{R(d_2),1}}), (\overline{u_{R(d_1),1}} \vee \overline{u_{R(d_2),1}}), \\ (\overline{u_{W(d_1),2}} \vee \overline{u_{W(d_2),2}}), (\overline{u_{W(d_1),2}} \vee \overline{u_{R(d_1),2}}), (\overline{u_{W(d_1),2}} \vee \overline{u_{R(d_2),2}}), (\overline{u_{W(d_2),2}} \vee \overline{u_{R(d_1),2}}), (\overline{u_{W(d_2),2}} \vee \overline{u_{R(d_2),2}}), (\overline{u_{R(d_1),2}} \vee \overline{u_{R(d_2),2}}), \\ (\overline{u_{W(d_1),3}} \vee \overline{u_{W(d_2),3}}), (\overline{u_{W(d_1),3}} \vee \overline{u_{R(d_1),3}}), (\overline{u_{W(d_1),3}} \vee \overline{u_{R(d_2),3}}), (\overline{u_{W(d_2),3}} \vee \overline{u_{R(d_1),3}}), (\overline{u_{W(d_2),3}} \vee \overline{u_{R(d_2),3}}), (\overline{u_{R(d_1),3}} \vee \overline{u_{R(d_2),3}}), \\ (\overline{u_{W(d_1),4}} \vee \overline{u_{W(d_2),4}}), (\overline{u_{W(d_1),4}} \vee \overline{u_{R(d_1),4}}), (\overline{u_{W(d_1),4}} \vee \overline{u_{R(d_2),4}}), (\overline{u_{W(d_2),4}} \vee \overline{u_{R(d_1),4}}), (\overline{u_{W(d_2),4}} \vee \overline{u_{R(d_2),4}}), (\overline{u_{R(d_1),4}} \vee \overline{u_{R(d_2),4}}) \end{array} \right\}$			
$C_3 \equiv \left\{ \begin{array}{l} (\overline{u_{R(d_2),2}} \vee \overline{u_{R(d_1),1}}), (\overline{u_{R(d_2),3}} \vee \overline{u_{R(d_1),2}} \vee \overline{u_{R(d_1),1}}), (\overline{u_{R(d_2),4}} \vee \overline{u_{R(d_1),3}} \vee \overline{u_{R(d_1),2}} \vee \overline{u_{R(d_1),1}}) \end{array} \right\}$			
$C_4 \equiv \left\{ \begin{array}{l} (\overline{u_{R(d_1),2}} \vee \overline{u_{W(d_1),1}}), (\overline{u_{R(d_1),3}} \vee \overline{u_{W(d_1),2}}), (\overline{u_{R(d_1),4}} \vee \overline{u_{W(d_1),3}}), (\overline{u_{R(d_2),2}} \vee \overline{u_{W(d_2),1}}), (\overline{u_{R(d_2),3}} \vee \overline{u_{W(d_2),2}}), (\overline{u_{R(d_2),4}} \vee \overline{u_{W(d_2),3}}) \end{array} \right\}$			
$C_5 \equiv \left\{ \begin{array}{l} (u_{W(d_2),4} \vee u_{R(d_2),4}) \end{array} \right\}$			

Figure 8.1: SAT Instance Generated From Example VMC Instance.

It is apparent that using such a transformation is not an efficient way to solve the VMC problem. Even for small VMC instances, the corresponding SAT instances are quite large.

We can optimize the transformation to reduce the size of the corresponding SAT instance. First, the fact that a schedule must respect program order enables us to rule out a number of possibilities. Second, a preprocessing step can scan the histories, removing reads that return values read/written by the previous operation in the same history ($O(n)$), and possibly eliminating some dead writes ($O(n^2)$). Third, once the preprocessing is done, we know that every read has at least one operation from another process history sched-

uled just before it. This information can be used to further constrain the range of positions in which a memory operation may reside in a schedule. Figure 8.2 shows the simple example from Figure 8.1, and the corresponding (optimized) SAT instance.

<p>VMC:</p> $H \equiv \{h_1, h_2, h_3\}$ $D \equiv \{d_1, d_2, d_1\}$ $d_r = d_2$	$\frac{h_1}{\underline{\underline{W(d_1)}}}$	$\frac{h_2}{\underline{\underline{W(d_2)}}}$	$\frac{h_3}{\underline{\underline{R(d_1)}}}$ $\underline{\underline{R(d_2)}}$
<p>SAT:</p> $U \equiv \left\{ \begin{array}{l} u_{W(d_1),1}, u_{W(d_1),2}, u_{W(d_1),3}, u_{W(d_1),4}, u_{W(d_2),1}, u_{W(d_2),2}, u_{W(d_2),3}, u_{W(d_2),4}, \\ u_{R(d_1),1}, u_{R(d_1),2}, u_{R(d_1),3}, u_{R(d_1),4}, u_{R(d_2),1}, u_{R(d_2),2}, u_{R(d_2),3}, u_{R(d_2),4} \end{array} \right\}$ $C \equiv C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5$ $C_1 \equiv \left\{ \begin{array}{l} (u_{W(d_1),1} \vee u_{W(d_1),2} \vee u_{W(d_1),3} \vee u_{W(d_1),4}), (u_{W(d_2),1} \vee u_{W(d_2),2} \vee u_{W(d_2),3} \vee u_{W(d_2),4}), \\ (u_{R(d_1),1} \vee u_{R(d_1),2} \vee u_{R(d_1),3}), (u_{R(d_2),3} \vee u_{R(d_2),4}) \end{array} \right\}$ $C_2 \equiv \left\{ \begin{array}{l} (\overline{u_{W(d_1),1} \vee u_{W(d_2),1}}), \\ (\overline{u_{W(d_1),2} \vee u_{W(d_2),2}}), (\overline{u_{W(d_1),2} \vee u_{R(d_1),2}}), (\overline{u_{W(d_2),2} \vee u_{R(d_1),2}}), \\ (\overline{u_{W(d_1),3} \vee u_{W(d_2),3}}), (\overline{u_{W(d_1),3} \vee u_{R(d_2),3}}), (\overline{u_{W(d_2),3} \vee u_{R(d_1),3}}), (\overline{u_{W(d_2),3} \vee u_{R(d_2),3}}), \\ (\overline{u_{W(d_1),4} \vee u_{W(d_2),4}}), (\overline{u_{W(d_1),4} \vee u_{R(d_2),4}}), (\overline{u_{W(d_2),4} \vee u_{R(d_2),4}}) \end{array} \right\}$ $C_3 \equiv \left\{ (\overline{u_{R(d_2),3} \vee u_{R(d_1),2} \vee u_{R(d_1),1}}), (\overline{u_{R(d_2),4} \vee u_{R(d_1),3} \vee u_{R(d_1),2} \vee u_{R(d_1),1}}) \right\}$ $C_4 \equiv \left\{ (\overline{u_{R(d_1),2} \vee u_{W(d_1),1}}), (\overline{u_{R(d_2),4} \vee u_{W(d_2),3}}) \right\}$ $C_5 \equiv \left\{ (u_{W(d_2),4} \vee u_{R(d_2),4}) \right\}$			

Figure 8.2: Optimized SAT Instance Generated From Example VMC Instance

Despite optimization, the size of the SAT instance improved by only constant factors. We still have n^2 variables, and $O(n^3)$ clauses. Overall, it does not appear that transforming the problem to SAT is a practical solution.

9 Future Work

A few open problems remain in the area of verifying memory coherence. The complexity of verifying memory coherence for the case of only two memory operations per process is unknown. In addition, the case for read-modify-writes where values are written at most twice remains an open problem. It might also be useful to obtain tighter bounds for the cases with polynomial time algorithms, if possible.

In the area of verifying memory consistency, much work remains to be done for memory consistency models with relaxed ordering and atomicity requirements. Though consistency models that provide memory coherence can be regarded as NP-Complete to verify, the precise conditions under which each of them remains NP-Complete are not well understood. Some memory consistency models will be easier to verify adherence to than other models, which may be of interest to designers of future systems.

Our work and the previous work in this area have only begun to explore the impact of practical restrictions and additional information on the complexity of the problem. For example, the impact of accessing individual bytes or words within a larger granularity of coherence has not been explored. We have sidestepped verification issues in handling misaligned memory accesses. Implicit constraints, resulting from knowledge of protocol implementations or program behavior may also simplify the problem. Some notion of logical time in the process histories may also reduce the number of possible ways to interleave them.

In this work, we have only considered the *offline* version of the problem, that is, when all the memory operations and their data values have been provided for an execution. For online error detection (*dynamic verification*), not all of this information is available. In fact, practical implementations will restrict us to a very limited view of processor and memory system activities, and severely limit the time and physical resources available for checking. Beyond efficient algorithms for the offline problems studied here, an important avenue for future research is the efficient dynamic detection of violations of memory coherence and consistency.

10 Conclusions

There is an increasing need for design verification and fault-tolerance in shared-memory multiprocessor systems, particularly in the subsystems that enforce memory coherence and consistency. To continue building reliable systems, we will need practical methods to determine whether these systems are providing coherence and consistency.

However, verifying coherence and consistency are inherently difficult problems. The results in this paper, along with those of the previous work [3,4,5,6], suggest that practical methods do not exist for verifying coherence and consistency without additional information from the hardware.

Furthermore, though verifying coherence may itself be of practical utility, it does not necessarily simplify the problem of verifying consistency. Consistency remains difficult to verify despite additional information that makes verifying coherence tractable. Thus, the problem of verifying consistency should be attacked directly.

11 References

- [1] D. Siewiorek and R. Swarz. “Reliable Computer Systems, Design and Evaluation”, (3rd ed), Natick, MA. A. K. Peters, 1998: 79-219.
- [2] The IBM e-server pSeries 690, “Reliability, Availability, Serviceability (RAS)”. Technical White Paper, IBM, Sep. 2001.
- [3] P. Gibbons and E. Korach. “On Testing Cache-Coherent Shared Memories, Proceedings of the 6th ACM Symposium on Parallel Algorithms and Architectures”, 1994:177-188.
- [4] P. Gibbons and E. Korach. “The Complexity of Sequential Consistency”. Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing, 1992:317-325.
- [5] P. Gibbons and E. Korach. “Testing Shared Memories”. SIAM Journal of Computing, Aug. 1997:1208-1244.
- [6] P. Gibbons and E. Korach. “New Results on the Complexity of Sequential Consistency”, Technical Report, AT&T Bell Laboratories, Murray Hill, NJ. Sep. 1993.
- [7] C. Tang, “Cache System Design in the Tightly Coupled Multiprocessor System”. Proceedings of the AFIPS National Computer Conference, 1976: 749-753.
- [8] L. Censier and P. Feautrier. “A New Solution to Coherence Problems in Multi-cache Systems”. IEEE Transactions on Computers, 27:12 (1978).
- [9] D. Abts, D. Lilja, A. Bataineh, and S. Scott. “Dimensions of Verifying the Hardware-Software Interface in a Shared-Memory Multiprocessor”, Laboratory for Advanced Research in Computing Technology and Compilers, Technical Report No. ARCTiC 99-04, 1999.
- [10] J. Hennessy and D. Patterson. “Computer Architecture: A Quantitative Approach” (2nd ed), San Francisco, CA. Morgan Kaufmann Publishers, 1996: 655-657.
- [11] D. Culler, J. Singh, and A. Gupta. “Parallel Computer Architecture: A Hardware/Software Approach”, San Francisco, CA. Morgan Kaufmann Publishers, 1998:273-277, 715.

- [12] Y. Afek, G. Brown, and M. Merritt. “Lazy Caching”. ACM Transactions on Programming Languages and Systems, 1993.
- [13] W. Collier, “Reasoning About Parallel Architectures”, Prentice Hall, 1992:19
- [14] K. Gharachorloo, “Memory Consistency Models for Shared-Memory Multiprocessors”. WRL Research Report 95/9. 1995.
- [15] DEC STD 032-0, “VAX Architecture Standard”. Digital Equipment Corporation. 1989.
- [16] R. Sites. “Alpha Architecture Reference Manual”. Digital Press, 1992.
- [17] L. Lamport. “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs”. IEEE Transactions on Computers, C28(9): 690-691, Sep. 1979.
- [18] S. Adve and K. Gharachorloo. “Shared Memory Consistency Models: A Tutorial”. IEEE Computer, 29(12): 66-76, Dec. 1996.
- [19] R. Alur, K. McMillan, and D. Peled. “Model-checking of Correctness Conditions for Concurrent Objects”. Proceedings of the 11th Symposium on Logic in Computer Science, IEEE, 1996:219-228.
- [20] A. Condon, A. Hu. “Automatable Verification of Sequential Consistency”. Proceedings of the 13th Symposium on Parallel Algorithms and Architectures. ACM, 2001: 113-121.
- [21] C. Papadimitriou. “The Theory of Database Concurrency Control“, Computer Science Press Inc. 1986: 31-42.
- [22] R. Taylor. “Complexity of Analyzing the Synchronization Structure of Concurrent Programs”, Acta Informatica 19, 1983: 57-84.
- [23] M. Garey and D. Johnson. “Computers and Intractability: A Guide to the Theory of NP-Completeness”. New York, W.H. Freeman and Company, 1979: 38-39, 95-107, 259.
- [24] P. Keleher, A. Cox, and W. Zwaenepoel. “Lazy Release Consistency for Software Distributed Shared Memory”. Proceedings of the 19th Annual International Symposium on Computer Architecture, 1992.

- [25] J. Gu, P. Purdom, J. Franco, and B. Wah. “Algorithms for the Satisfiability (SAT) Problem: A Survey”. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1991.
- [26] I. Gent and T. Walsh. “The Search for Satisfaction”, Department of Computer Science, University of Strathclyde, Glasgow, Scotland. 1999.
- [27] S. Cook. “The Complexity of Theorem-Proving Procedures”, Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York: 151-158.
- [28] R. Karp. “Reducibility Among Combinatorial Problems”, in R. Miller and J. Thatcher (eds.) Complexity of Computer Computations, New York, Plenum Press: 85-103.
- [29] C. Papadimitriou and K. Steiglitz. “Combinatorial Optimization: Algorithms and Complexity” New York, Dover Publications Inc. 1982.

Appendix

Following is additional information on NP-Completeness and Propositional Satisfiability. The interested reader should consult Garey and Johnson's book [23], or Stephen Cook's original paper [27] for more complete and authoritative explanations.

A.1 Complexity and NP

In terms of algorithmic complexity, a problem is considered *tractable* if the solution can be computed in time polynomially related to the parameters of the problem (*polynomial-time*). For example, a simple sorting routine could have an upper bound on its run-time that is proportional to the square of the size of the input set. These problems make up the class P , or *deterministic polynomial*.

A superset of the class P is NP (*nondeterministic polynomial*), consisting of problems for which a solution can be *verified* in polynomial-time. In addition to the problems in P , NP contains many important problems in mathematics, science, and engineering for which polynomial-time algorithms have not been found.

It would be advantageous to find polynomial-time algorithms for all of the problems of NP , meaning that $P=NP$. Alternatively, if we could prove that $P \neq NP$, then we could conclude that some of the problems in NP are *intractable*, and stop looking for polynomial-time algorithms for them. Unfortunately, whether or not $P=NP$ is unknown.

In his landmark paper, Cook proved that all problems in NP could be *reduced* to the problem of *propositional satisfiability* (SAT) in polynomial-time [27]. For an instance of any other problem in NP , a polynomial-time algorithm can transform it to an analogous instance of SAT. This was a powerful result, since it meant that finding a polynomial-time algorithm for SAT would yield polynomial-time algorithms for all the other problems in NP .

Later, Karp proved that SAT could be reduced to a collection of 21 other NP problems (and therefore these problems are also at least as hard as any other in NP) [28]. A polynomial-time algorithm for any one these problems could be used as a component

in polynomial-time algorithms for all of the others, and similarly, a proof of intractability of any one applies to all the others. These problems came to be known as the class *NP-Complete*, and SAT came to be known as the first NP-Complete problem. Presently, there are thousands of NP-Complete problems, and the set continues to grow.

Since it is not known whether $P=NP$, we cannot be certain that a polynomial-time algorithm does not exist for the NP-Complete problems (though it is considered highly unlikely). Still, a proof of *NP-Completeness* is a common way to show a problem is difficult, because a solution would be a tremendous breakthrough. The question of whether $P=NP$ is one of the biggest open questions in mathematics and computer science.

A.2 Decision Problems and Promise Problems

When reasoning about the complexity of a problem, we usually state the problem in the form of a *decision problem*. A decision problem consists of a statement of the information available in an instance, and a question that can be answered with a “yes” or “no”. An algorithm for a decision problem would attempt to answer the question for a given instance based on the information available. For example, a simple decision problem can be stated as follows:

DEFINITION A.1: DIVISIBILITY

INSTANCE: Two finite numbers, a and b ; $b > 0$.

QUESTION: Is there an integer q such that $q*b = a$?

We are given two numbers, and asked find an integer by which we can multiply the second number to get the first. If there is, then the first number is divisible by the second. An algorithm for this problem might divide the first number by the second, and return “yes” if the result is an integer or the remainder is zero. This algorithm would be trivially polynomial in its runtime.

Not all problems can be stated as decision problems. A superset of the decision problems is the set of *promise problems*. For these problems, a promise is made about the

nature of the input. We can reason about the complexity of the problem using the promise as a restriction, but then our results only apply to the instances for which the promise is kept (for which the restriction holds). We may want to define a promise problem to reason about a restricted variant of a problem for which verifying that an instance satisfies the restriction is itself a difficult problem. A simple example is given below.

DEFINITION A.2: TRAVELING SALESPERSON with HAMILTONIAN PATH

INSTANCE: A graph $G = \{V, E\}$ with finite set V of vertices, finite set E of edges $\{v_1, v_2\} \subseteq V$, cost function $C: E \rightarrow Z^+$, and finite maximum cost K .

PROMISE: G has a Hamiltonian Path: a directed path $E' \subseteq E$ visiting each node $v \in V$ exactly once.

QUESTION: Is there a directed path $E' \subseteq E$ visiting each node exactly once, for which the total cost $\sum C(e)$ for all edges $e \in E'$ is less than K ?

This problem differs from the typically stated traveling salesperson problem in that we are now promised that a path exists. We must still determine if there is an affordable path. Unfortunately, there is not an efficient way to determine if a given graph satisfies this promise, since finding a Hamiltonian Path is a known NP-Complete problem! A polynomial-time algorithm for this promise problem (if one exists) would have to assume a path exists, since there is no way to check for one in polynomial time. Its behavior would be undefined for other types of graphs.

A.3 Proving NP-Completeness

First, the problem must be a member of NP. Second, and perhaps most important, a known NP-Complete problem must reduce to the new problem in polynomial time.

To prove membership in NP, we must show there is a concise way of stating a solution to an instance such that it can be verified in polynomial time. This is called a certificate [29]. For example, a certificate for a SAT instance would be a truth assignment for the variables, with which we could simply check for an unsatisfied clause. The word concise is a bit vague, but keep in mind that a computer cannot inspect and verify a

solution in polynomial time if it cannot be stated in space polynomially related to the size of the problem.

The last part proves the problem is NP-Hard (at least as hard as any other problem in NP), and we say that a decision problem is NP-Complete if it is both NP-Hard and a member of NP. Proving NP-Hardness is usually a little more difficult than proving membership in NP. Let Π be a known NP-Complete problem, and Π' be the new problem. We must find a transformation that, given an arbitrary instance I of Π , converts it into an instance I' of Π' , and can be implemented to run in polynomial time. This must be done carefully such that a solution to I' can be easily converted back into a solution for I . Since this is done for an arbitrary instance, it follows that a polynomial-time algorithm for Π' could be used to solve Π . The new problem is therefore at least as hard as an NP-Complete problem, and by transitivity, at least as hard as all problems in NP.

A.4 Propositional Satisfiability

In mathematical logic, *Propositional Satisfiability* refers to a decision problem where the instance is a Boolean expression (usually expressed in conjunctive normal form), and the question is whether there is a truth assignment for the variables such that the expression evaluates to *True*. The problem is best stated by Garey and Johnson as follows:

Let $U = \{u_1, u_2, \dots, u_m\}$ be a set of Boolean variables. A truth assignment for U is a function $t: U \rightarrow \{T, F\}$. If $t(u) = T$ we say that u is “true” under t ; if $t(u) = F$ we say that u is “false.” If u is a variable in U , then u and \bar{u} are *literals* over U . The literal u is true under t if and only if the variable u is true under t ; the literal \bar{u} is true if and only if the variable u is false.

A *clause* over U is a set of literals over U , such as $\{u_1, \bar{u}_3, u_8\}$. It represents the disjunction of those literals and is *satisfied* by a truth assignment if and only if at least one of its members is true under that assignment. The clause above will be satisfied by t unless $t(u_1) = F$, $t(u_3) = T$, and $t(u_8) = F$. A collection C of clauses over

U is *satisfiable* if and only if there exists some truth assignment for U that simultaneously satisfies all the clauses in C . Such a truth assignment is called a *satisfying truth assignment* for C .

SATISFIABILITY

INSTANCE: A set U of variables and a collection C of clauses over U .

QUESTION: Is there a satisfying truth assignment for C ?

For example, $U = \{u_1, u_2\}$ and $C = \{\{u_1, \bar{u}_2\}, \{\bar{u}_1, u_2\}\}$ provide an instance of SAT for which the answer is “yes.” A satisfying truth assignment is given by $t(u_1) = t(u_2) = T$. On the other hand, replacing C by $C' = \{\{u_1, u_2\}, \{u_1, \bar{u}_2\}, \{\bar{u}_1\}\}$ yields an instance for which the answer is “no”... [23]

In addition to the general problem stated above, there are a number of important variants of SAT. For example, there is 3SAT, in which each clause has exactly three literals [23]. Other examples include NOT-ALL-EQUAL 3SAT, and ONE-IN-THREE 3SAT [23].

Many important problems in science and engineering can be stated as SAT problems, so efficient algorithms for SAT would be of great practical utility. However, as mentioned in previous sections, SAT is in general NP-Complete. Further, many of the variants, such as 3SAT, are also NP-Complete.

There are special cases in which the problem can be solved efficiently (e.g., if each clause has less than three literals [23]). For many practical applications, the common case does not require an exponential search of the solution space. Furthermore, research in heuristics, randomization, approximation, and parallelization have made it possible to tackle SAT problems with thousands of variables [25,26].