The Complexity of Verifying Memory Coherence and Consistency

Jason F. Cantin, Mikko H. Lipasti, James E. Smith

Department of Electrical and Computer Engineering

University of Wisconsin-Madison

1415 Engineering Drive

Madison, WI 53706

{ jcantin, mikko, jes }@ece.wisc.edu

September 18th, 2004

Index Terms

B.3.2.g [**Hardware**]: Memory Structures: Design Styles---*Shared memory*; B.3.4.b [**Hardware**]: Reliability, Testing, and Fault-Tolerance---*Error-checking*; F.2.2.f [**Theory of Computation**] Nonnumerical Algorithms and Problems---*Sequencing and Scheduling*.

Abstract

We study the problem of testing shared memories for violations of memory coherence and consistency. We first prove that detecting violations of coherence in an execution is NP-Complete, and show that it remains NP-Complete for a number of very restricted instances. We then use this result to prove that all known consistency models are NP-Hard to verify. Finally, we show that the complexity of verifying consistency models is not a mere consequence of coherence, and that verifying sequential consistency remains NP-Complete once coherence has been verified.

1 Introduction

Memory coherence and consistency are important features of a shared-memory multiprocessor system. Memory coherence requires that operations to a shared location appear to execute in a serial order. Memory consistency is defined by memory consistency models, which specify serialization requirements for all memory operations. Together with the instruction set, memory coherence and consistency form the hardware/software interface of shared-memory multiprocessor systems.

It is becoming increasingly difficult to design and test modern shared-memory multiprocessor systems, with design complexity increasing to achieve higher levels of performance. Current systems incorporate cache hierarchies, multiple networks, distributed memory controllers, and various protocol optimizations to improve performance. To make matters worse, shrinking transistor dimensions and rising power dissipation in digital integrated circuits are increasing the susceptibility of hardware to errors. Yet, there is a lack of work addressing the problem of practically testing shared memories for violations of coherence and consistency and the problem of detecting protocol hardware errors. Traditional approaches have focused only on the detection of data corruption and computation errors [1, 2].

We are researching techniques to detect violations of memory coherence and consistency dynamically, to develop techniques that could be used for hardware error detection or in simulations to evaluate and debug memory system designs. This paper presents a theoretical investigation into the complexity of the problem, carried out as a first step. First, "verifying memory coherence" for an execution is stated formally as a decision problem (VMC). The problem is then proven NP-Complete, and analyzed under a number of important restrictions to characterize the problem. We find VMC to be NP-Complete for executions with as few as three memory operations per process and at most two writes of each data value, or executions with as few as two read-modify-write operations per process and values written by at most three read-modify-write operations.

All memory consistency models either provide memory coherence, or allow the programmer to force a serialization with special instructions. Therefore, the NP-Completeness of VMC directly implies the NP-Hardness of verifying adherence to a memory consistency model. Finally, we show that the complexity of verifying adherence to a memory consistency

model is not a mere consequence of requiring coherence, because verifying sequential consistency remains NP-Complete once coherence has been verified.

The rest of this paper is organized as follows. Important related work is discussed in Section 2. Section 3 defines important terminology and relations used in the rest of this paper. Section 4 proves that verifying coherence is NP-Complete with a reduction from SAT. Section 5 presents a collection of results for restricted cases of verifying memory coherence. Section 6 extends the complexity results for coherence to memory consistency models, and presents a proof that verifying sequential consistency is NP-Complete for coherent executions. This is followed by a discussion of future work in Section 7. Finally, Section 8 concludes the paper.

2 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 Verifying Sequential Consistency (VSC) and Verifying Linearizability (VL) problems, and proved they were NP-Complete. They presented a collection of results for restricted cases, characterizing the complexity of the problems.

Our interpretation of memory coherence is equivalent to sequential consistency when 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 was not fully explored, and the general case was left as an open problem in [6]. This paper addresses this problem, with new results that apply to all consistency models.

Very recently, Gontmakher, Polyakov, and Schuster analyzed the complexity of verifying *Java consistency* (the Java memory model) [7]. This work is similar to that of Gibbons and Korach on sequential consistency, except the Java consistency model is weaker and relaxes many of the ordering requirements. However, because this model requires memory coherence, the NP-Hardness follows trivially from the results in this paper. This paper shall be the final word on the complexity of testing shared memory consistency.

Alur, McMillan, and Peled studied the problems of verifying that a system provides serializability, linearizability, and sequential consistency, and proved that these problems are in PSPACE, in EXPSPACE, and undecidable for arbitrary protocols, respectively [8]. They accomplished this by showing that you can encode the halting problem for an arbitrary two-counter automaton as the sequential consistency of a finite state machine. The key to the reduction is the clever way to hide the unbounded state of the counters in the unbounded amount of logical time by which processors can be out of sync, while still being sequentially consistent. Condon and Hu identified a restricted (and more realistic) class of protocols for which verifying sequential consistency is decidable in [9], and together with Bingham improved on this work with a broader class of protocols in [10]. Qadeer developed a decidable model-checking technique based on the properties of causality and data independence that practical protocols typically satisfy [11]. This requires that the logical order of write events to a location match the temporal order, which is true of the write-invalidate protocols predominantly used today. The decidability of verifying that memory coherence is maintained by a protocol remains unknown.

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 [12]. 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 this problem is a schedule of the operations, and the task is to show that the schedule yields the same results as a serial schedule.

Taylor studied synchronization in concurrent programs using the *rendezvous* mechanism [13]. 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 has similarities to our problem. An analogous question might be "Is it possible for a read to be mapped to a particular write?" However, not all write operations need to be observed in an execution 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. This is similar in that we need to determine the order in which to execute jobs on a finite resource, based on certain dependences. However, many write operations might write the value needed by a read, but a precedence relation need only hold for one in a schedule. Much of this work is summarized in [14].

3 Preliminaries

Similar to previous work, memory read operations are of the form "R(a, d)", and write operations are of the form "W(a, d)". The address of the operation is represented by *a*, and *d* represents the data read/written by the operation. When all memory operations have the same

address, we use the shorthand notation "R(d)", and "W(d)". Atomic read-modify-write operations are denoted "RW(a, d_r, d_w)", or simply "RW(d_r, d_w)"; where d_r is the data read and d_w is the data written. For simplicity, assume that all memory operations are aligned word accesses.

A *process history* is a sequence of memory operations from the execution of a process, in program order, including the values read/written by each operation. Here, the memory operations in a process history will be written vertically, from top to bottom in program order.

Prior to program execution, every location in a shared memory is in some state. We refer to this as the *initial value* of the location, and denote it $d_I[a]$. Similarly, every memory location is in some state after a program has executed, and we refer to this as the *final value* ($d_F[a]$).

A *coherent schedule* is an interleaving of single-address process histories, where every read operation returns the value written by the immediately preceding write operation (the exception being reads that precede the first write, which must return the initial value d_I). The last write in the schedule must write the final value for the location. A multiprocessor execution is considered *coherent* if a coherent schedule exists for each address.

A *sequentially consistent schedule* is a schedule of the memory operations from an execution that demonstrates sequential consistency [15] was provided. In other words, a schedule of all the memory operations (from all addresses), in which every read returns the value written by the immediately preceding write with the same address. This is equivalent to the *legal schedule* used in [3, 4, 5, 6].

4 Verifying Memory Coherence

To reason about the complexity of verifying memory coherence, we first define a new decision problem "Verifying Memory Coherence" (VMC).

DEFINITION 4.1: Verifying Memory Coherence

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

QUESTION: Is there a coherent schedule S for the operations of H with address a?

The VMC problem is NP-Complete. This can be proven with a reduction from the seminal NP-Complete problem SAT [14]. Given an instance Q of SAT with variable set U and collection of clauses C, we construct an instance V of VMC such that V has a coherent schedule S if and only if Q has satisfying truth assignment T.

The key idea is that two unique data values (d_u and $d_{\bar{u}}$) can encode the truth assignment T of each variable u in U. The truth of u corresponds to the order in which these values are written in a schedule S (4.1). We create two process histories h_1 and h_2, each writing one of these data values for each variable u, so that their interleaving sets T for U. The literals for each variable u (u and $\bar{u}$) are represented by two process histories (h_u and $h_{\bar{u}}$), which each read the data values in the order that corresponds to *true* for the literal. Once h_1 and h_2 have been interleaved into a schedule S, only the process histories representing literals *true* under T may be included in S.

$$T : U \mapsto \{True, False\}$$

$$W(d_u) \xrightarrow{S} W(d_{\bar{u}}) \quad \Leftrightarrow \quad T(u) = True \tag{4.1}$$

$$W(d_{\bar{u}}) \xrightarrow{S} W(d_u) \quad \Leftrightarrow \quad T(\bar{u}) = True$$

A clause is satisfied by a truth assignment if at least one of its literals is *true* under the assignment. All clauses in C must be simultaneously satisfied under some truth assignment T for the instance Q to be satisfiable. For each clause c in C a write of a special value d_c is appended to each process history $h_u/h_{\bar{u}}$ for each literal that appears in c, such that d_c may be written in S only if c is satisfied under T.

Another process history h_3 is defined with read operations that return each value d_c. This process history may be added to S only if each value has been written (i.e., T satisfies C). After all the reads in h_3, we append a second set of writes of the values d_u and $d_{\bar{u}}$ that represent each variable u. If T satisfies C, we can include h_3 in S, and schedule these writes so that the unique data values appear in both orders in a schedule S, and the remaining process histories (for false-literals) may be included in S.

The complete reduction is illustrated in Figure 4.1, with a general SAT instance in the top portion and the corresponding VMC instance in the bottom portion. For a SAT instance with m variables and n clauses, the corresponding VMC instance has $2m+3$ process histories and $O(mn)$ operations. It is easy to see that this reduction may be performed in polynomial time.

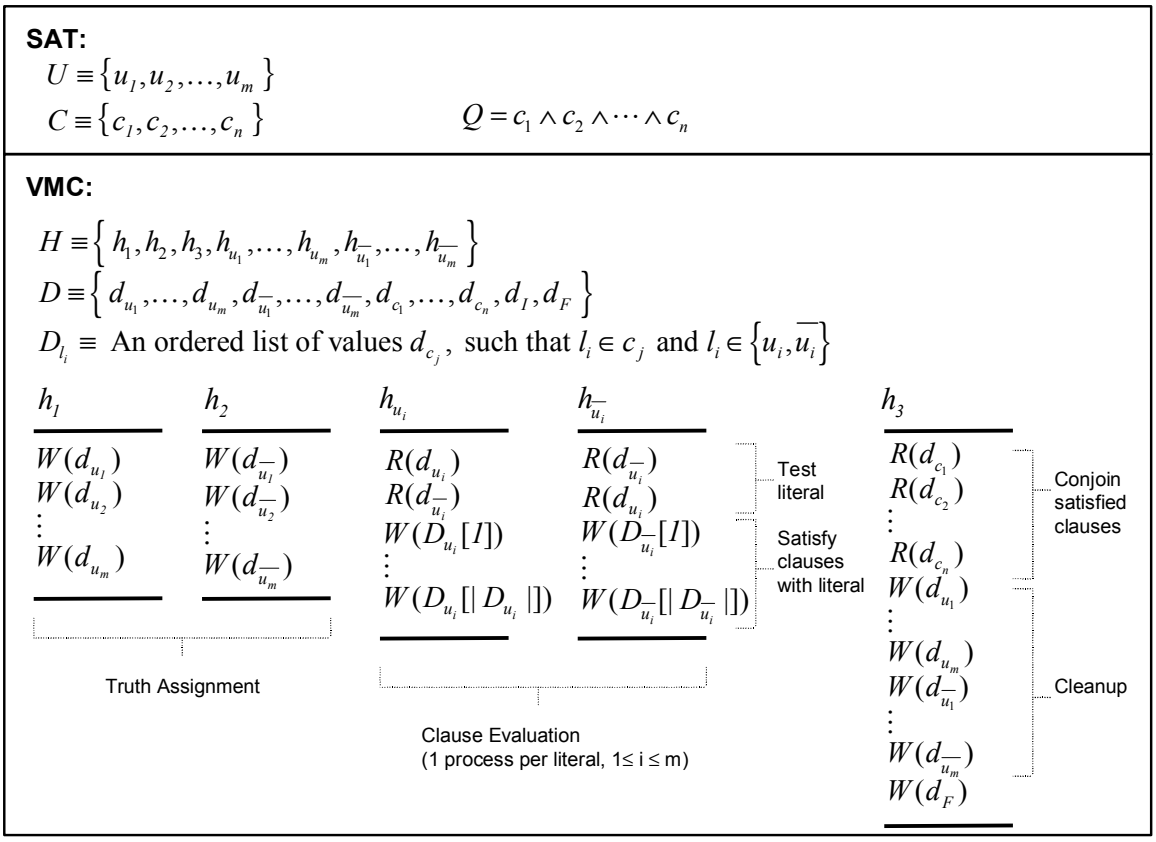

Figure 4.1: General SAT to VMC Reduction

As an example, consider the VMC instance depicted in Figure 4.2. This instance corresponds to the SAT instance $Q=u$, with one variable, u, and a unit clause c consisting of the literal u. The reader may verify that a coherent schedule may be constructed if and only if the write operation $W(d_u)$ precedes the write operation $W(d_{\bar{u}})$ in that schedule.

SAT:

$U \equiv \{ u \}$

$C \equiv \{ c \}$

$Q = c = u$

VMC:

$H \equiv \{ h_1, h_2, h_3, h_u, h_{\bar{u}} \}$

$D \equiv \{ d_u, d_{\bar{u}}, d_{c_1} \}$

h_1	h_2	h_u	$h_{\bar{u}}$	h_3
$W(d_u)$	$W(d_{\bar{u}})$	$R(d_u)$	$R(d_{\bar{u}})$	$R(d_c)$
		$R(d_{\bar{u}})$	$R(d_u)$	$W(d_u)$
		$W(d_c)$		$W(d_{\bar{u}})$

Figure 4.2: Example VMC Instance for SAT Instance $Q = u$

THEOREM 4.2: VMC is NP-Complete

Proof:

1. Membership in NP:

A certificate for VMC is a coherent schedule of the memory operations. We can easily determine whether a given schedule S is a coherent schedule for a given instance V of VMC. We first scan S to see if it contains all the memory operations from V, in their respective program orders. While scanning, we keep track the last 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 VMC instance from Figure 4.1.

LEMMA 4.3: 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 $W(d_u)$ and $W(d_{\bar{u}})$ from h_1 and h_2 appear in S in some order. This corresponds to a truth assignment T for U.

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 it. However, a write operation $W(d_c)$ only appears in the process histories that represent literals that appear in c. 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. Besides h_3, the only other process histories that write the values read by this process history 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, the truth assignment T is such that for every clause c in C, at least one of the literals in c is *true* under T. Therefore, T satisfies C and 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. 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 $R(d_c)$ for every clause c before the write operations. Each may be paired with one of the writes $W(d_c)$. Hence, it is possible to construct a schedule S that includes h_3.

The write operations from h_3 can provide data for 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, a coherent schedule S exists, and V is coherent.

It follows that there is a coherent schedule S for H if and only if Q is satisfiable. Since VMC is in NP, and SAT reduces to VMC in polynomial time, VMC is NP-Complete. □

5 Restricted / Augmented Cases of Verifying Memory Coherence

In this section we discuss the complexity of the VMC problem under a number of restrictions. Specifically, we restrict the number of memory operations per process, and the number of times a data value is written. We also consider executions in which only read-modify-write operations are allowed (it was observed that under some circumstances all memory operations may be treated as read-modify-writes [6]). In addition, we consider the case in which the memory system has been augmented to provide the order in which write operations were executed.

The VMC problem is equivalent to the VSC problem [3, 4, 5, 6] for executions that use only one shared variable. Consequently, some results for restricted cases of VMC were already obtained by, or logically follow from results obtained in previous work. In the interest of space, we only present reductions for new results, and summarize all known results at the end.

5.1 Restricted Cases

We find that the VMC problem remains NP-Complete for as few as three simple operations (reads and writes) per process, and data values written at most twice. Figure 5.1 depicts a reduction from 3SAT [14] to a VMC instance meeting these constraints. The VMC problem is also NP-Complete with only two read-modify-write operations per process (previously known [5]) and data values written at most three times (Figure 5.2).

It follows from previous work that the VMC problem is tractable if every process history has only one operation, or data values are written at most once (i.e., the read-map is known) [5]. The problem is also tractable when the number of process histories is restricted to a constant number. With n total memory operations, k process histories and c addresses, the VSC problem can be solved in $O(n^k k^c)$ time [6]. All instances of VMC are instances of VSC in which $c=1$. Thus, the complexity of verifying coherence for n total operations and k process histories is $O(kn^k)$, which is polynomial for constant k. Similarly, instances of VMC with only read-modify-write operations and a constant number of process histories are instances of the corresponding VSC problem with only one location, which has $O(n^k)$ time complexity [6]. Therefore, VMC with n total read-modify-write operations and k process histories has $O(n^k)$ time complexity.

The case for VMC with only two simple memory operations per process remains an open problem. In addition, the time complexity of the case with only read-modify-write operations and data values written at most twice is also unknown.

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

$$D \equiv \left\{ d_{u_1},\ldots,d_{u_m}, d_{\overline{u_1}},\ldots,d_{\overline{u_m}}, d_{c_1,1},\ldots,d_{c_1,3},\ldots,d_{c_n,3} \right\}$$

$D_{u_i} \equiv$ An ordered list of values $d_{c_j,k}$ such that u_i is the k^{th} literal of c_j

$D_{\overline{u_i}} \equiv$ An ordered list of values $d_{c_j,k}$ such that $\overline{u_i}$ is the k^{th} literal of c_j

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

Figure 5.1 3SAT to VMC, 3 Memory Operations Per Process, Values Written At Most Twice

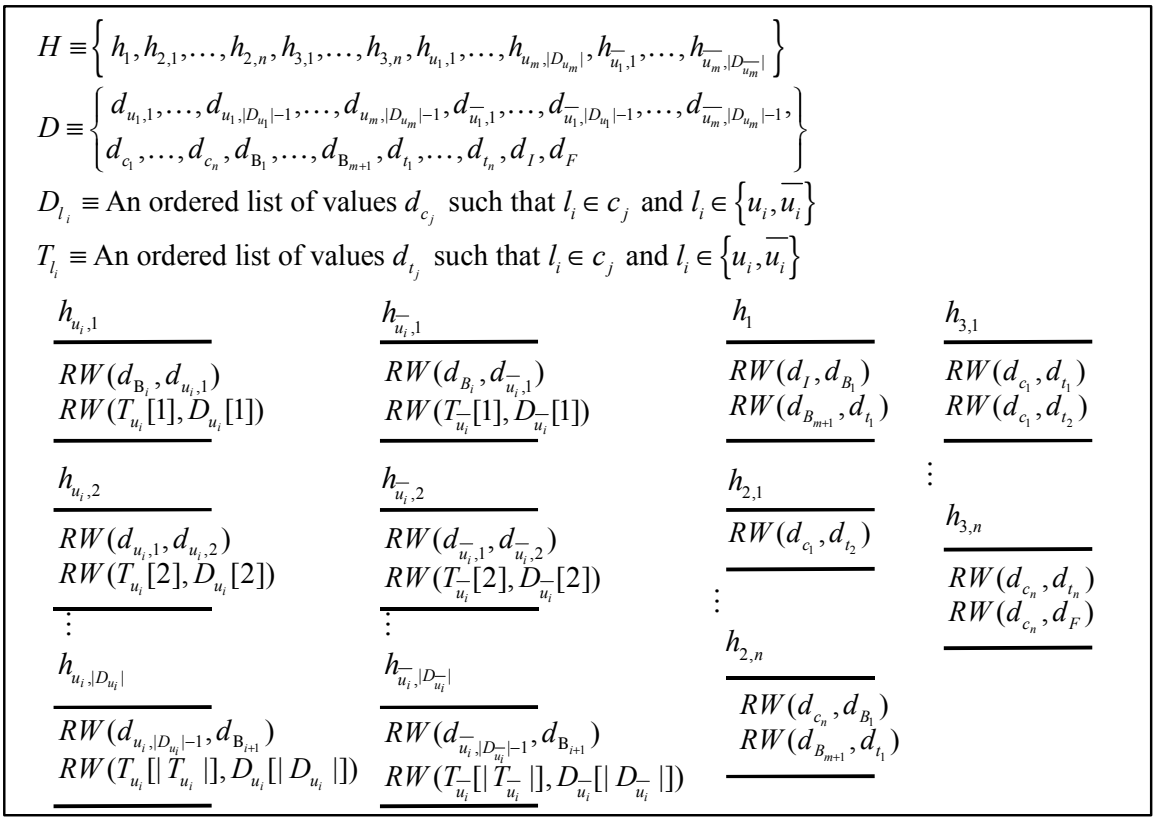

Figure 5.2 3SAT to VMC, Two Read-Modify-Writes, Values Written At Most Three Times

5.2 Supplying the Order of Writes

We find that VMC becomes tractable if the memory system is provides the order in which write operations were executed (i.e., the *write-order* [3, 4, 5, 6]). VMC has an $O(n^2)$ time algorithm for n total operations, and an $O(n)$ time algorithm if all the operations read-modify-writes.

To construct a schedule, we use the write-order as a starting point, and try to insert the read operations. For each read x, we first check to see if the data value of the last operation from the same process history y matches that of x, and if so insert x right after y in the schedule. Otherwise, we scan forward from y to the next write operation from the same process history (if x is the first operation from the history, we begin scanning from the beginning of the schedule

and the initial value). If while scanning, a write of the same data value is found, we insert *x* right after it. Note that in the special case where all operations are read-modify-write operations, the write-order is in fact a total order, and we can simply check that the read component of each read-modify-write returns the value of the write-component of the preceding read-modify-write.

5.3 Summary

The complexity results for verifying memory coherence are summarized in Figure 5.3. Here, *n* denotes the total number of memory operations and *k* the number of process histories. The shaded entries indicate new results. Cases for which the complexity is not known have a question mark in the corresponding table entry. For simplicity we have listed the restrictions individually, though the problem remains NP-Complete when restricting both the number of operations per process history and the number of writes of each value, as shown above. Other results were either obtained in, or follow from, previous work.

	Simple Reads/Writes	Read-Modify-Writes
1 Operation/Process	O(nlg(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 (Read-map)	O(n)	O(nlg(n))
2 Writes/Value	NP-Complete	**?**
3+ Writes/Value	See above	NP-Complete
Write-order Given	$O(n^2)$	O(n)

Figure 5.3: Summary of Complexity Results for Memory Coherence.

6 Verifying Memory Consistency

The NP-Completeness of verifying coherence implies the NP-Hardness of verifying adherence to memory consistency models, including sequential consistency. However, verifying coherence first does not necessarily simplify the problem of verifying consistency. We also show that verifying sequential consistency remains NP-Complete for executions that are coherent.

6.1 Sequential Consistency

To verify that sequential consistency was maintained during an execution, we can find a sequentially consistent schedule for all the memory operations. This decision problem (6.1) was defined and proven to be NP-Complete in [3, 4, 5, 6]. Note that the VSC problem, as defined, is only useful for reasoning about sequential consistency (other models are a different problem).

DEFINITION 6.1: Verifying Sequential Consistency (VSC)

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.

QUESTION: Is there a sequentially consistent schedule S for H?

Though the complexity of VSC is already known from previous work, it is interesting to observe that the complexity of VSC also follows from our results with coherence. Because every instance of VMC is an instance of VSC, the NP-Hardness of VMC implies the NP-Hardness of VSC. It is easy to see that VSC is in the class NP, so the proof follows by restriction.

6.2 Other Consistency Models

In addition to sequential consistency, many other consistency models exist, most of which have more relaxed ordering requirements for performance. These include the models from Sun: Total Store Order (TSO), Partial Store Order (PSO), and Relaxed Memory Ordering (RMO); models implemented by Intel: Processor Consistency (PC), and Release Consistency (RC); IBM models such as the PowerPC model [16]; the Alpha model; and academic models such as CRF [17], just to name a few. A fairly complete list of hardware-implemented models with references can be found in [18].

All of the hardware-implemented memory consistency models in the literature reduce to memory coherence for executions that access only one shared location [18]. For these models, verifying consistency is at least as difficult as verifying coherence, and hence they are NP-Hard to verify. However, it remains to be shown whether verifying adherence to these consistency models is in NP. A new decision problem would be needed for each model, with a consistent schedule defined under that model. In some cases, consistent schedules are not straightforward to define and may be too awkward to work with, necessitating the use of dependence graphs.

The reductions presented here do not directly apply to consistency models that relax the coherence requirement (e.g., Lazy Release Consistency [19]). However, these consistency models (like other weak models) provide special instructions with which the programmer can override such relaxations when necessary. We can therefore extend our reductions to these models by using these instructions to enforce a serial order for some address. For LRC, we modify the reduction in Figure 4.1 by placing *acquire* and *release* operations around each memory operation (Figure 6.1). As long as memory operations to some address must appear

serialized, either by implicit consistency model requirements or explicit synchronization, the reductions presented here apply.

It is worth noting that memory consistency models can be quite arbitrarily defined, and some may relax the coherence requirement without providing the programmer with primitives for explicit synchronization. However, software for such a consistency model would be extremely difficult if not impossible to develop, and no implementations are known to exist.

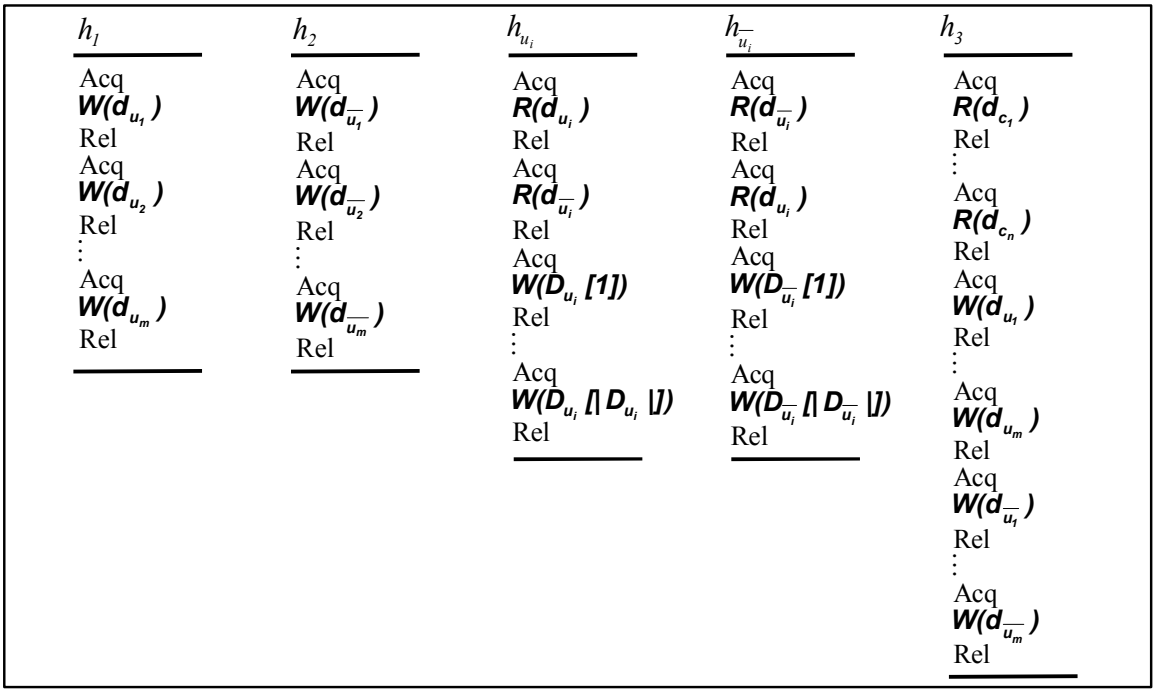

Figure 6.1 VMC Instance of Figure 4.1, with Synchronization

6.3 Consistency with Coherence

Verifying coherence does not necessarily simplify the problem of verifying consistency. It is NP-Complete to verify that sequential consistency was provided for an execution even when it is known that coherence was provided.

The problem of verifying sequential consistency for executions that are coherent is defined below. Since there is currently not an efficient way to verify that an arbitrary instance is coherent, this is not a decision problem but rather a promise problem.

DEFINITION 6.2: 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 a in A, there exists a coherent schedule.

QUESTION: Is there a sequentially consistent schedule S for H?

A SAT instance Q with m variables and n clauses may be reduced to a VSCC instance V with $2m+3$ processes and $m+n+1$ shared locations. The reduction is very similar to the one used for VMC. A unique address a_u is used to represent the truth assignment of a variable u. The truth of u corresponds to the order in which two values (d_X and d_Y) are written to a_u in a schedule S (6.1). Two process histories (h_u and $h_{\bar{u}}$) are defined for each variable u to represent the literals, each reading the values d_X and d_Y in the order that corresponds to *true* for the literal. Once the values have been written, only the process histories representing literals that are *true* under the corresponding truth assignment may be included in S.

$$T : U \mapsto \{True, False\}$$

$$\begin{aligned} W(a_u, d_X) \xrightarrow{S} W(a_u, d_Y) \quad &\Leftrightarrow \quad T(u) = True \\ W(a_u, d_Y) \xrightarrow{S} W(a_u, d_X) \quad &\Leftrightarrow \quad T(\bar{u}) = True \end{aligned} \tag{6.1}$$

For each clause c satisfied by a literal, a write operation of value d_Z to a special location with address a_c is appended to the process history that represents the literal. Another process history, h_3, is defined with read operations that return the value d_Z from each location a_c. This history may be included in a schedule S only if each of these locations has been written.

After a special location (a_Δ) is written at the end of h_3, the locations corresponding to each variable are rewritten by h_1 and h_2. This allows the values to appear in both orders in a schedule S, so that the process histories corresponding to literals that are *false* under the assignment may be included. The reduction is summarized in Figure 6.2, the full proof may be found in [20].

$H \equiv \left\{ h_1, h_2, h_3, h_{u_1}, \ldots, h_{u_m}, h_{\overline{u_1}}, \ldots, h_{\overline{u_m}} \right\}$

$D \equiv \{ d_X, d_Y, d_Z, d_I \}$

$A \equiv \left\{ a_{u_1}, \ldots, a_{u_m}, a_{c_1}, \ldots, a_{c_n}, a_\Delta \right\}$

$A_{u_i} \equiv$ An ordered list of values a_{c_j} such that $u_i \in c_j$

$A_{\overline{u_i}} \equiv$ An ordered list of values a_{c_j} such that $\overline{u_i} \in c_j$

h_1	h_2	h_{u_i}	$h_{\overline{u_i}}$	h_3
$W(a_{u_1}, d_X)$	$W(a_{u_1}, d_Y)$	$R(a_{u_i}, d_X)$	$R(a_{u_i}, d_Y)$	$R(a_{c_1}, d_Z)$
$\vdots$	$\vdots$	$R(a_{u_i}, d_Y)$	$R(a_{u_i}, d_X)$	$R(a_{c_2}, d_Z)$
$W(a_{u_m}, d_X)$	$W(a_{u_m}, d_Y)$	$W(A_{u_i}[1], d_Z)$	$W(A_{\overline{u_i}}[1], d_Z)$	$\vdots$
$R(a_\Delta, d_Z)$	$R(a_\Delta, d_Z)$	$\vdots$	$\vdots$	$R(a_{c_n}, d_Z)$
$W(a_{u_1}, d_Y)$	$W(a_{u_1}, d_X)$	$W(A_{u_i}[\lvert A_{u_i} \rvert], d_Z)$	$W(A_{\overline{u_i}}[\lvert A_{\overline{u_i}} \rvert], d_Z)$	$W(a_\Delta, d_Z)$
$\vdots$	$\vdots$			
$W(a_{u_m}, d_Y)$	$W(a_{u_m}, d_X)$			

Figure 6.2 SAT to VSCC

The memory operations of the constructed VSCC instance are separated by address in Figure 6.3. A coherent schedule is indicated for each address. Depending on the address, we have one of three cases. The first case (top) consists of the memory operations used to assign the truth of a

variable. We can construct a coherent schedule by interleaving the uncomplemented literal's history with h_1, interleaving the complemented literal's history with h_2, and concatenate the resulting schedules. In the other cases (middle and bottom), only one value is written to the location, so we may trivially schedule all the reads after all the writes. Thus, the instance is coherent by construction.

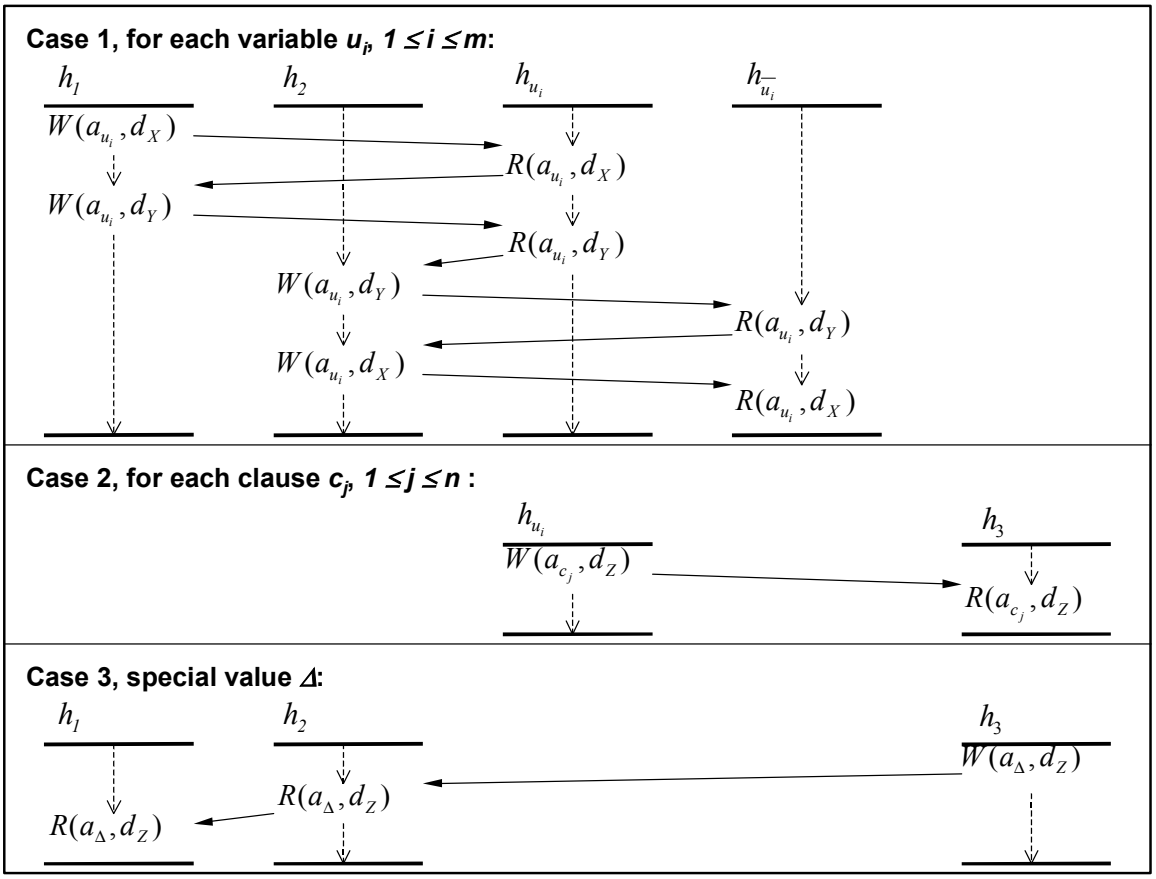

Figure 6.3 VSCC Instance, Separated By Address

Furthermore, we can constrain the problem such that coherence may be efficiently verified. Recall that VMC is in P when the order in which write operations were executed is provided. With the write-order, the VSCC problem is reduced to a decision problem in which we can

verify coherence in polynomial time. 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, VSCC not only remains NP-Complete for coherent executions, but also when information is provided to efficiently verify coherence.

Going still further, we can use the schedules constructed while verifying coherence as input to the VSC problem. Encoded in a coherent schedule is a serial order for all the write operations, and a mapping from the read operations to write operations. It was shown previously that this information can be used to generate a sequentially consistent schedule in $O(n \lg n)$ 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 one set that can be merged into a sequentially consistent schedule. Hence, the failure to find a sequentially consistent schedule may only mean that the wrong set of coherent schedules was produced when verifying coherence. Like all NP-Complete problems, VSC is resistant to divide-and-conquer approaches.

7 Future Work

In this paper we have only dealt with the problem of testing shared memories for coherence. An important question is whether we can verify that a protocol maintains coherence beforehand. While it has been shown that verifying protocols maintain sequential consistency can be undecidable, the complexity (or even decidability) of verifying that a protocol maintains coherence remains unknown. This would be an important result with implications for all memory consistency models and the protocols that implement them.

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.

8 Concluding Remarks

Verifying memory 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 significant additional information from the system. Practical offline verification with software or online error detection with hardware will be difficult to implement.

Further, though verifying coherence may itself be of practical utility, doing so does not necessarily simplify the problem of verifying consistency. Consistency remains difficult to verify despite additional information that makes verifying coherence tractable.

Acknowledgements

This research was supported by NSF grant CCR-0083126, IBM, and fellowships from the NSF and the UW Foundation. We are very thankful to Dieter van Melkebeek for vetting all our proofs, for proofreading drafts, and for many valuable discussions on complexity theory. We are also grateful for the comments and suggestions of Deborah Joseph, Eric Bach, Paramjit Oberoi, Shiliang Hu, Kevin Lepak, and Carl Mauer.

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] A. Gontmakher, S. Polyakov, and A. Schuster. "Complexity of Verifying Java Shared Memory Execution". Parallel Processing Letters, World Scientific Publishing Company.

[8] R. Alur, K. McMillan, and D. Peled. "Model-checking of Correctness Conditions for Concurrent Objects". Proc. 11th Symp. Logic in Computer Science, IEEE, 1996:219-228.

[9] A. Condon, A. Hu. "Automatable Verification of Sequential Consistency". Symposium on Parallel Algorithms and Architectures. ACM, 2001: 113-121.

[10] J. Bingham, A. Condon, and A. Hu. "Toward a Decidable Notion of Sequential Consistency". Proc. 15th ACM Symp. on Parallel Algorithms and Architectures, 2003.

[11] S. Qadeer. “Verifying Sequential Consistency on Shared-Memory Multiprocessors by Model-Checking”. IEEE Trans. Parallel and Distributed Systems, Vol. 14, No. 8, 2003.

[12] C. Papadimitriou. “The Theory of Database Concurrency Control“, Computer Science Press Inc. 1986: 31-42.

[13] R. Taylor. “Complexity of Analyzing the Synchronization Structure of Concurrent Programs”, Acta Informatica 19, 1983: 57-84.

[14] M. Garey and D. Johnson. “Computers and Intractability: A Guide to the Theory of NP-Completeness”. NY, W.H. Freeman and Company, 1979: 38-39, 95-107, 259

[15] L. Lamport. “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs”. IEEE Transactions on Computers, C28(9): 690-691, Sep. 1979.

[16] C. May, E. Silha, R. Simpson, and H. Warren (Eds). “The PowerPC Architecture: A Specification for a New Family of RISC Processors” (2nd ed). Morgan Kaufmann Publishers, Inc., CA, 1994.

[17] X. Shen, Arvind, and L. Rudolph. “Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers”. ISCA 2000.

[18] K. Gharachorloo, “Memory Consistency Models for Shared-Memory Multiprocessors”. WRL Research Report 95/9. 1995.

[19] P. Keleher, A. Cox, and W. Zwaenepoel. “Lazy Release Consistency for Software Distributed Shared Memory”. International Symposium on Computer Architecture, 1992.

[20] J. Cantin. “The Complexity of Verifying Memory Coherence”. Tech. Report ECE-03-01, University of Wisconsin-Madison, 2003.