The Complexity of Verifying Memory Coherence and Consistency

Jason F. Cantin, *Student Member*, *IEEE*, Mikko H. Lipasti, *Member*, *IEEE*, and James E. Smith, *Member*, *IEEE*

Abstract—The problem of testing shared memories for memory coherence and consistency is studied. First, it is proved that detecting violations of coherence in an execution is NP-Complete, and it remains NP-Complete for a number of restricted instances. This result leads to a proof that all known consistency models are NP-Hard to verify. The complexity of verifying consistency models is not a mere consequence of coherence, and verifying sequential consistency remains NP-Complete even after coherence has been verified.

Index Terms—Hardware, memory structures, design styles, shared memory, reliability, testing, fault-tolerance, error-checking, theory of computation, nonnumerical algorithms and problems, sequencing and scheduling.

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, consistency forms 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. Traditional approaches have focused only on the detection of data corruption and computation errors [1], [2].

The research described in this paper is directed at techniques that detect violations of memory coherence and consistency. Such techniques can be useful for hardware error detection, and evaluation and debugging of memory system designs. The results reported here serve as a first step and are part of a theoretical investigation into the complexity of the problem. *Verifying Memory Coherence* (VMC) for a given program execution is stated formally as a decision problem. The problem is proved to be NP-Complete and is analyzed under a number of important restrictions to characterize the problem. VMC is shown to be NP-Complete 1) for executions with as few as three memory operations per process and with at most two writes of each data value, and 2) for executions with as few as two read-modify-write operations per process and with values written by at most three read-modify-write operations.

All memory consistency models implemented in practice either provide memory coherence or allow the programmer to serialize memory operations with special instructions. Therefore, the NP-Completeness of VMC directly implies the NP-Hardness of verifying adherence to a memory consistency model. Furthermore, 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, followed by important terminology and relations in Section 3. Section 4 proves that verifying coherence is NP-Complete with a reduction from SAT, followed by a collection of results for restricted cases of verifying memory in Section 5. 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 problems for future work in Section 7. Finally, Section 8 concludes the paper.

2 Related Work

The most relevant related work is that of Gibbons and Korach on analyzing the complexity of detecting violations of sequential consistency and linearizability [3], [4], [5], [6]. They define the *Verifying Sequential Consistency* (VSC) and *Verifying Linearizability* (VL) problems, prove they are NP-Complete, and present a collection of results characterizing their complexity.

Our interpretation of memory coherence is equivalent to sequential consistency when restricted to one location, so some results of Gibbons and Korach also apply to memory coherence. However, the complexity of the general problem of verifying sequential consistency for one location was left

• *The authors are with the Department of Electrical and Computer Engineering, University of Wisconsin-Madison, 3652 Engineering Hall, 1415 Engineering Drive, Madison, WI 53706.*
E-mail: {jcantin, mikko, jes}@ece.wisc.edu.

Manuscript received 29 Apr. 2003; revised 9 Mar. 2004; accepted 6 Oct. 2004; published online 20 May 2005.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0066-0403.

as an open problem in [6]. This paper addresses this problem, with new results that apply to all consistency models.

Very recently, Gontmakher et al. analyzed the complexity of verifying *Java consistency* (the Java memory model) [7]. This work is similar to that of Gibbons and Korach, except the Java consistency model has weaker ordering requirements. However, because this model requires memory coherence, the NP-Hardness follows trivially from the results presented in this paper.

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

Papadimitriou [12] studies concurrency control for database transactions and proves that verifying view-serializability is NP-Complete. 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 [13] studies synchronization in concurrent programs using the *rendezvous* mechanism. 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 the problem studied in this paper. 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.

3 Preliminaries

Similar to previous work [3], [4], [5], [6], 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 is the data read/written by the operation. When all memory operations have the same address, the shorthand notation is "R(d)," and "W(d)." Atomic read-modify-write operations are denoted "$\mathrm{RW}(\mathrm{a}, \mathrm{d_r}, \mathrm{d_w})$," or simply "$\mathrm{RW}(\mathrm{d_r}, \mathrm{d_w})$," where d_r is the data read and d_w is the data written. For simplicity, all operations are assumed to be 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, the *initial value* of the location, denoted as $d_I[a]$. Similarly, the state after the program has executed is 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 (except 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] is satisfied. In other words, a schedule of all the memory operations, in which every read returns the value written by the immediately preceding write with the same address. This is equivalent to the *legal schedule* defined in [3], [4], [5], [6].

4 Verifying Memory Coherence

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

Definition 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 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 $d_{\overline{u}}$) 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 (1). Two process histories h_1 and h_2 are created, each writing one of the data values for each variable u, so that their interleaving establishes T for U. The literals for each variable u (u and $\overline{u}$) are represented by two process histories (h_u and $h_{\overline{u}}$) that each read the data values in the order that corresponds to *true* for the literal. Once h_1 and h_2 are interleaved into a schedule S, only the process histories representing literals *true* under T may then be included in S.

$$\begin{aligned} &T : U \mapsto \{True, False\} \\ &W(d_u) \overset{S}{\longrightarrow} W(d_{\overline{u}}) \iff T(u) = True \\ &W(d_{\overline{u}}) \overset{S}{\longrightarrow} W(d_u) \iff T(\overline{u}) = True. \end{aligned} \tag{1}$$

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

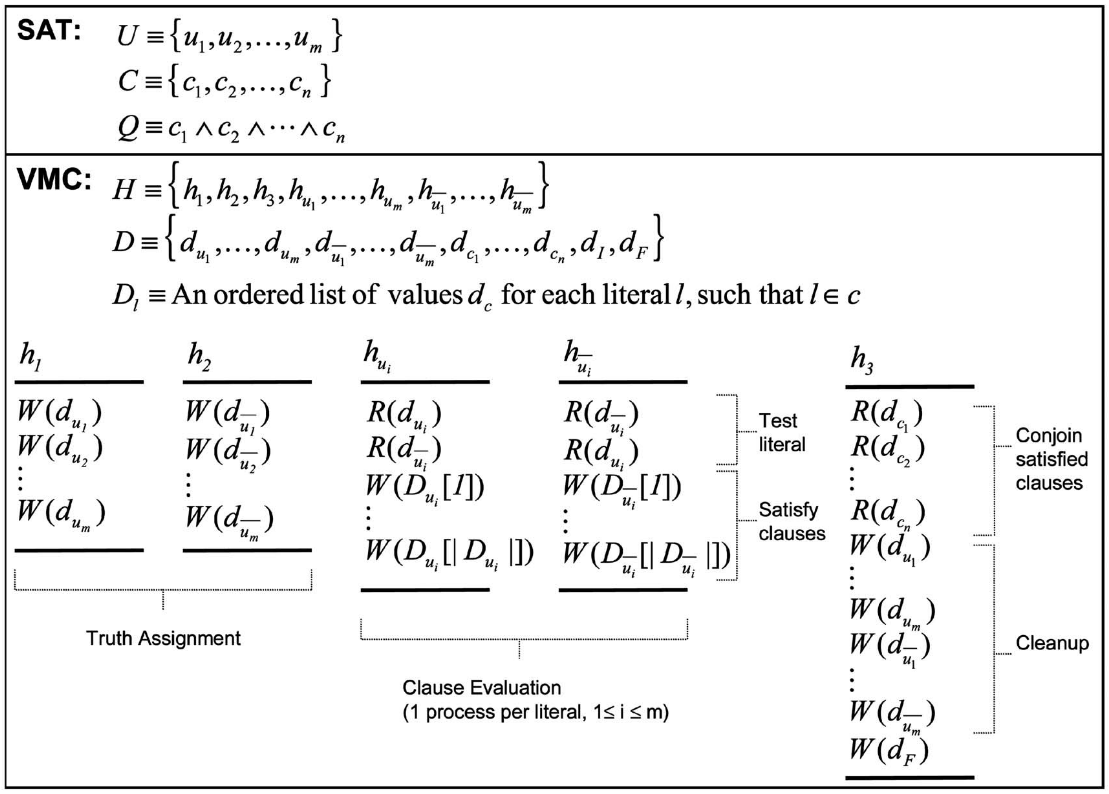

SAT:
$U \equiv \{u_1, u_2, \ldots, u_m\}$
$C \equiv \{c_1, c_2, \ldots, c_n\}$
$Q \equiv c_1 \wedge c_2 \wedge \cdots \wedge c_n$
VMC:
$H \equiv \{h_1, h_2, h_3, h_{u_1}, \ldots, h_{u_m}, h_{\bar{u}_1}, \ldots, h_{\bar{u}_m}\}$
$D \equiv \{d_{u_1}, \ldots, d_{u_m}, d_{\bar{u}_1}, \ldots, d_{\bar{u}_m}, d_{c_1}, \ldots, d_{c_n}, d_I, d_F\}$
$D_l \equiv$ An ordered list of values d_c for each literal l, such that $l \in c$
h_1: $W(d_{u_1})$, $W(d_{u_2})$, ⋮, $W(d_{u_m})$
h_2: $W(d_{\bar{u}_1})$, $W(d_{\bar{u}_2})$, ⋮, $W(d_{\bar{u}_m})$
h_{u_i}: $R(d_{u_i})$, $R(d_{\bar{u}_i})$, $W(D_{u_i}[1])$, ⋮, $W(D_{u_i}[|D_{u_i}|])$
$h_{\bar{u}_i}$: $R(d_{\bar{u}_i})$, $R(d_{u_i})$, $W(D_{\bar{u}_i}[1])$, ⋮, $W(D_{\bar{u}_i}[|D_{\bar{u}_i}|])$
h_3: $R(d_{c_1})$, $R(d_{c_2})$, ⋮, $R(d_{c_n})$, $W(d_{u_1})$, ⋮, $W(d_{u_m})$, $W(d_{\bar{u}_1})$, ⋮, $W(d_{\bar{u}_m})$, $W(d_F)$
Test literal
Satisfy clauses
Conjoin satisfied clauses
Cleanup
Truth Assignment
Clause Evaluation
(1 process per literal, 1≤ i ≤ m)

Fig. 1. General SAT to VMC reduction.

in C a write of a special value d_c is appended to the process history $h_u/h_{\overline{u}}$ 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, a second set of writes of the values d_u and $d_{\overline{u}}$ is appended to represent each variable u. If T satisfies C, one 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, finally, the remaining process histories (for false-literals) may be included in S.

The complete reduction is illustrated in Fig. 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. This reduction may be performed in polynomial time.

As an example, consider the VMC instance depicted in Fig. 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 $W(d_u)$ from h_1 precedes the write operation $W(d_{\overline{u}})$ from h_2.

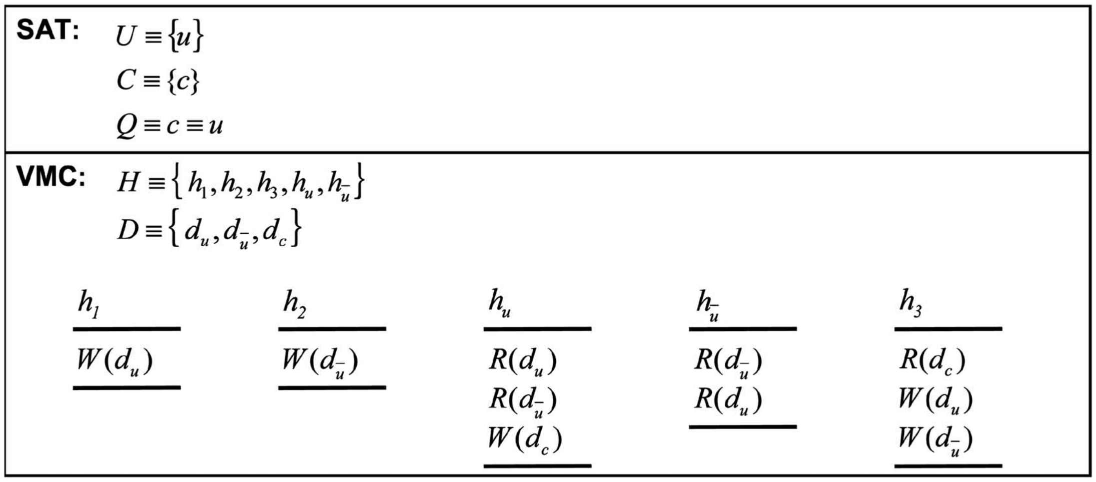

SAT:
$U \equiv \{u\}$
$C \equiv \{c\}$
$Q \equiv c \equiv u$
VMC:
$H \equiv \{h_1, h_2, h_3, h_u, h_{\bar{u}}\}$
$D \equiv \{d_u, d_{\bar{u}}, d_c\}$
h_1: $W(d_u)$
h_2: $W(d_{\bar{u}})$
h_u: $R(d_u)$, $R(d_{\bar{u}})$, $W(d_c)$
$h_{\bar{u}}$: $R(d_{\bar{u}})$, $R(d_u)$
h_3: $R(d_c)$, $W(d_u)$, $W(d_{\bar{u}})$

Fig. 2. Example VMC instance for SAT instance $Q = u$.

Theorem 1. *VMC is NP-Complete.*

Proof.

1. Membership in NP: A certificate for VMC is a coherent schedule of the memory operations. One can easily determine whether a given schedule S is a coherent schedule for a given instance V of VMC. The schedule S is scanned to determine if it contains all the memory operations from V, in their respective program orders. While scanning, the last value written is tracked to 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 Fig. 1. □

Lemma 1. *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 $W(d_{\overline{u}})$ from h_1 and h_2 appear in S in some order. This encodes 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 representing the literals are h_1 and h_2, and these write each value only once. This means that h_1 and h_2 must be 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 is *true* under 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 (1). 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 clause 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, 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 in h_3. Each may be paired with one of the writes $W(d_c)$ from a process history representing a literal that is *true* under T. 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.

It follows that V is coherent 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, the number of memory operations per process and the number of times a data value is written are restricted. 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 the previous work. In the interest of space, we present only reductions for new results, and summarize all known results at the end.

5.1 Restricted Cases

We show 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. Fig. 3 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 (Fig. 4).

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. The time complexity of the case with only read-modify-write operations and data values written at most twice is also unknown.

$$H \equiv \left\{ \begin{array}{l} h_{1,1},\ldots,h_{1,\lceil m/3\rceil},h_{2,1},\ldots,h_{2,\lceil m/3\rceil},h_{3,1,1},\ldots,h_{3,3,1},\ldots,h_{3,3,n}, \\ h_{4,1},\ldots,h_{4,m},h_{u_1,1},\ldots,h_{u_m,\lvert D_{u_m}\rvert},h_{\bar{u}_1,1},\ldots,h_{\bar{u}_m,\lvert D_{\bar{u}_m}\rvert} \end{array} \right\}$$

$$D \equiv \left\{ d_{u_1},\ldots,d_{u_m},d_{\bar{u}_1},\ldots,d_{\bar{u}_m},d_{c_1,1},\ldots,d_{c_1,3},\ldots d_{c_n,3} \right\}$$

$D_l \equiv$ An ordered list of values $d_{c,k}$ for each literal l, such that l is the k^{th} literal of c

$h_{1,1}$	$h_{2,1}$	$h_{u_i,1}$	$h_{\bar{u}_i,1}$	$h_{3,1,1}$	$h_{3,2,1}$	$h_{3,3,1}$	$h_{4,1}$
$W(d_{u_1})$	$W(d_{\bar{u}_1})$	$R(d_{u_i})$	$R(d_{\bar{u}_i})$	$R(d_{c_1,1})$	$R(d_{c_1,2})$	$R(d_{c_1,3})$	$R(d_{c_n,1})$
$W(d_{u_2})$	$W(d_{\bar{u}_2})$	$R(d_{\bar{u}_i})$	$R(d_{u_i})$	$W(d_{c_1,2})$	$W(d_{c_1,3})$	$W(d_{c_1,1})$	$W(d_{u_1})$
$W(d_{u_3})$	$W(d_{\bar{u}_3})$	$W(D_{u_i}[1])$	$W(D_{\bar{u}_i}[1])$				$W(d_{\bar{u}_1})$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$h_{1,\lceil m/3\rceil}$	$h_{2,\lceil m/3\rceil}$	$h_{u_i,\lvert D_{u_i}\rvert}$	$h_{\bar{u}_i,\lvert D_{\bar{u}_i}\rvert}$	$h_{3,1,n}$	$h_{3,2,n}$	$h_{3,3,n}$	$h_{4,m}$
$W(d_{u_{m-2}})$	$W(d_{\bar{u}_{m-2}})$	$R(d_{u_i})$	$R(d_{\bar{u}_i})$	$R(d_{c_{n-1},1})$	$R(d_{c_n,2})$	$R(d_{c_n,3})$	$R(d_{c_n,1})$
$W(d_{u_{m-1}})$	$W(d_{\bar{u}_{m-1}})$	$R(d_{\bar{u}_i})$	$R(d_{u_i})$	$R(d_{c_n,1})$	$W(d_{c_n,3})$	$W(d_{c_n,1})$	$W(d_{u_m})$
$W(d_{u_m})$	$W(d_{\bar{u}_m})$	$W(D_{u_i}[\lvert D_{u_i}\rvert])$	$W(D_{\bar{u}_i}[\lvert D_{\bar{u}_i}\rvert])$	$W(d_{c_n,2})$			$W(d_{\bar{u}_m})$

Fig. 3. 3SAT to VMC, 3 memory operations per process, values written at most twice.

$$H \equiv \left\{ h_1,h_{2,1},\ldots,h_{2,n},h_{3,1},\ldots,h_{3,n},h_{u_1,1},\ldots,h_{u_m,\lvert D_{u_m}\rvert},h_{\bar{u}_1,1},\ldots,h_{\bar{u}_m,\lvert D_{\bar{u}_m}\rvert} \right\}$$

$$D \equiv \left\{ \begin{array}{l} d_{u_1,1},\ldots,d_{u_1,\lvert D_{u_1}\rvert-1},\ldots,d_{u_m,\lvert D_{u_m}\rvert-1},d_{\bar{u}_1,1},\ldots,d_{\bar{u}_1,\lvert D_{u_1}\rvert-1},\ldots,d_{\bar{u}_m,\lvert D_{u_m}\rvert-1}, \\ d_{c_1},\ldots,d_{c_n},d_{B_1},\ldots,d_{B_{m+1}},d_{c_1,2},\ldots,d_{c_n,2},d_I,d_F \end{array} \right\}$$

$D_l \equiv$ An ordered list of values d_c for each literal l, such that $l \in c$

$T_l \equiv$ An ordered list of values $d_{c,2}$ for each literal l, such that $l \in c$

$h_{u_i,1}$	$h_{\bar{u}_i,1}$	h_1	$h_{3,1}$
$RW(d_{B_i},d_{u_i,1})$	$RW(d_{B_i},d_{\bar{u}_i,1})$	$RW(d_I,d_{B_1})$	$RW(d_{c_1},d_{t_1})$
$RW(T_{u_i}[1],D_{u_i}[1])$	$RW(T_{\bar{u}_i}[1],D_{\bar{u}_i}[1])$	$RW(d_{B_{m+1}},d_{t_1})$	$RW(d_{c_1},d_{t_2})$
$h_{u_i,2}$	$h_{\bar{u}_i,2}$	$h_{2,1}$	⋮
$RW(d_{u_i,1},d_{u_i,2})$	$RW(d_{\bar{u}_i,1},d_{\bar{u}_i,2})$	$RW(d_{c_1},d_{t_2})$	$h_{3,n}$
$RW(T_{u_i}[2],D_{u_i}[2])$	$RW(T_{\bar{u}_i}[2],D_{\bar{u}_i}[2])$		$RW(d_{c_n},d_{t_n})$
⋮	⋮	⋮	$RW(d_{c_n},d_F)$
$h_{u_i,\lvert D_{u_i}\rvert}$	$h_{\bar{u}_i,\lvert D_{\bar{u}_i}\rvert}$	$h_{2,n}$	
$RW(d_{u_i,\lvert D_{u_i}\rvert-1},d_{B_{i+1}})$	$RW(d_{\bar{u}_i,\lvert D_{\bar{u}_i}\rvert-1},d_{B_{i+1}})$	$RW(d_{c_n},d_{B_1})$	
$RW(T_{u_i}[\lvert T_{u_i}\rvert],D_{u_i}[\lvert D_{u_i}\rvert])$	$RW(T_{\bar{u}_i}[\lvert T_{\bar{u}_i}\rvert],D_{\bar{u}_i}[\lvert D_{\bar{u}_i}\rvert])$	$RW(d_{B_{m+1}},d_{t_1})$	

Fig. 4. 3SAT to VMC, two read-modify-writes, values written at most three times.

5.2 Supplying the Order of Writes

VMC becomes tractable if the memory system 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 are read-modify-writes.

	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	NP-Complete	NP-Complete
Write-order Given	O(n^2)	O(n)

Fig. 5. Summary of complexity results for memory coherence.

To construct a schedule, the write-order is used as a starting point, and the read operations are inserted. For each read x, one should 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, scan forward in the schedule from y to the next write operation from the same process history (if x is the first operation from the history, begin scanning from the beginning of the schedule and the initial value). If, while scanning a write of the same data value is found, x is inserted immediately following 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 one 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 Fig. 5. 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 unknown have a question mark in the table. 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 obtained in, or follow from, previous work.

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 show that verifying sequential consistency remains NP-Complete when executions are coherent.

6.1 Sequential Consistency

To verify that sequential consistency is maintained during an execution, one can find a sequentially consistent schedule for all the memory operations. Definition 2 is defined and proved to be NP-Complete in [3], [4], [5], [6]. Note that, as defined, the VSC problem is only useful for reasoning about sequential consistency (other models are a different problem).

Definition 2: Verifying Sequential Consistency (VSC).

INSTANCE: Data value set D, address set A, and 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

Besides sequential consistency, many other consistency models exist, most of which have more relaxed ordering requirements for performance. These include models from SPARC [16]: Total Store Order (TSO), Partial Store Order (PSO), and Relaxed Memory Ordering (RMO); Intel x86 models [17]: Processor Consistency (PC), and Release Consistency (RC); IBM models such as the PowerPC model [18]; the Alpha model; and academic models such as CRF [19], just to name a few. A more complete list of models with references can be found in [20].

All of the hardware-implemented memory consistency models in the literature reduce to memory coherence for executions that access only one shared location [20]. 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.

The reductions presented earlier do not directly apply to consistency models that relax the coherence requirement (e.g., Lazy Release Consistency [21]). 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, the reduction in Fig. 1 is modified by placing *acquire* and *release* operations around each memory operation (Fig. 6). As long as memory operations to some address must appear serialized, either by implicit consistency model requirements or explicit program 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.

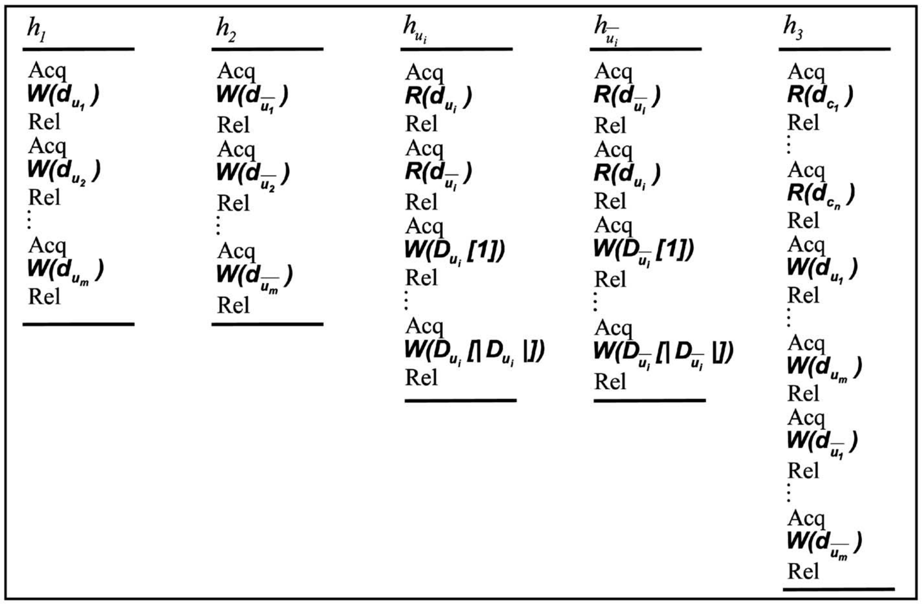
6.3 Consistency with Coherence

Verifying coherence does not necessarily simplify the problem of verifying consistency. It is NP-Complete to verify that sequential consistency is 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 no known efficient way to verify that an arbitrary instance is coherent, this is not a decision problem but rather a promise problem.

Definition 3: Verifying Sequential Consistency with Coherence (VSCC).

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



h_1	h_2	h_{u_i}	$h_{\overline{u}_i}$	h_3
Acq	Acq	Acq	Acq	Acq
$W(d_{u_1})$	$W(d_{\overline{u}_1})$	$R(d_{u_i})$	$R(d_{\overline{u}_i})$	$R(d_{c_1})$
Rel	Rel	Rel	Rel	Rel
Acq	Acq	Acq	Acq	⋮
$W(d_{u_2})$	$W(d_{\overline{u}_2})$	$R(d_{\overline{u}_i})$	$R(d_{u_i})$	Acq
Rel	Rel	Rel	Rel	$R(d_{c_n})$
⋮	⋮	Acq	Acq	Rel
Acq	Acq	$W(D_{u_i}[1])$	$W(D_{\overline{u}_i}[1])$	Acq
$W(d_{u_m})$	$W(d_{\overline{u}_m})$	Rel	Rel	$W(d_{u_1})$
Rel	Rel	⋮	⋮	Rel
		Acq	Acq	⋮
		$W(D_{u_i}[\lvert D_{u_i}\rvert])$	$W(D_{\overline{u}_i}[\lvert D_{\overline{u}_i}\rvert])$	Acq
		Rel	Rel	$W(d_{u_m})$
				Rel
				Acq
				$W(d_{\overline{u}_1})$
				Rel
				⋮
				Acq
				$W(d_{\overline{u}_m})$
				Rel

Fig. 6. VMC instance of Fig. 1, with synchronization.

PROMISE: For each address in A, there exists a coherent schedule.

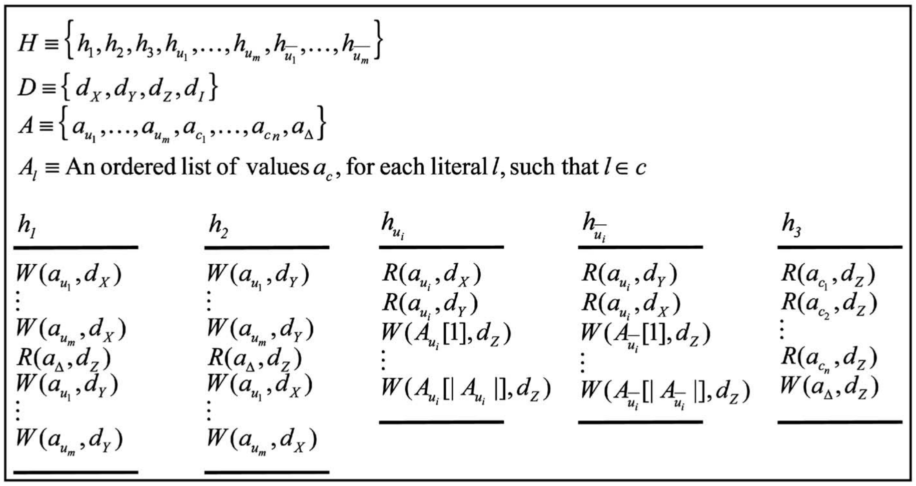
QUESTION: Is there a sequentially consistent schedule S for H?

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 (2). Two process histories (h_u and $h_{\overline{u}}$) 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.

$$\begin{aligned} &T : U \mapsto \{True, False\} \\ &W(a_u, d_X) \overset{s}{\longrightarrow} W(a_u, d_Y) \Longleftrightarrow T(u) = True \\ &W(a_u, d_Y) \overset{s}{\longrightarrow} W(a_u, d_X) \Longleftrightarrow T(\overline{u}) = True. \end{aligned} \tag{2}$$

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 process histories corresponding to literals that are *false* under the assignment to be included. The reduction is summarized in Fig. 7, the full proof may be found in [22].



$H \equiv \{h_1, h_2, h_3, h_{u_1}, \ldots, h_{u_m}, h_{\overline{u}_1}, \ldots, h_{\overline{u}_m}\}$

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

$A \equiv \{a_{u_1}, \ldots, a_{u_m}, a_{c_1}, \ldots, a_{cn}, a_\Delta\}$

$A_l \equiv$ An ordered list of values a_c, for each literal l, such that $l \in c$

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

Fig. 7. SAT to VSCC.

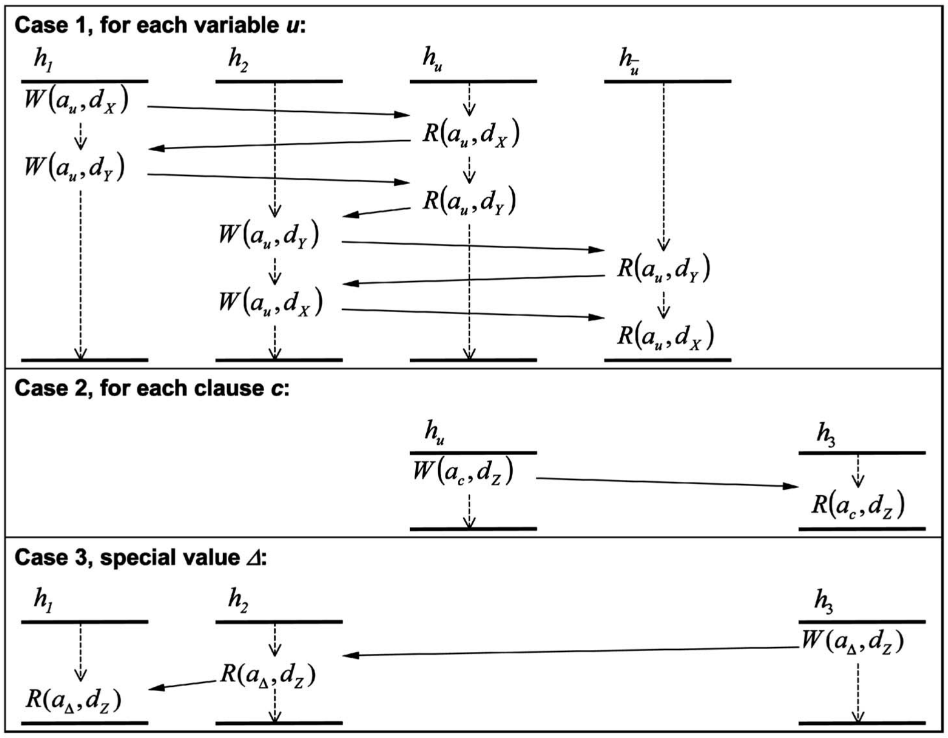


Fig. 8. VSCC instance, separated by address.

The memory operations of the constructed VSCC instance are separated by address in Fig. 8, with a coherent schedule indicated for each. Depending on the address, there is one of three cases. The first case (top) consists of the memory operations used to assign the truth of a variable. A coherent schedule can be constructed by interleaving the uncomplemented literal's history with h_1, interleaving the complemented literal's history with h_2, and concatenating the resulting schedules. In the other cases (middle and bottom), only one value is written to the location, so all the reads can be trivially scheduled after the writes. Thus, the instance is coherent by construction.

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 coherence can be verified in polynomial time. However, it is 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 were produced when verifying coherence. Like all NP-Complete problems, VSC is resistant to divide-and-conquer approaches.

7 Future Work

The work in this paper dealt only with the problem of testing shared memory systems for coherence. An important question is whether one can verify that a protocol maintains coherence beforehand. While it has been shown that verifying protocols maintain sequential consistency can be undecidable, the complexity of verifying that a protocol maintains coherence remains unknown. This would be an important result with implications for all consistency models.

Some 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, as well as the case for read-modify-writes where values are written at most twice.

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. Consistency remains difficult to verify despite additional information that makes verifying coherence tractable. Practical offline verification with software or online error detection with hardware will be difficult to implement.

ACKNOWLEDGMENTS

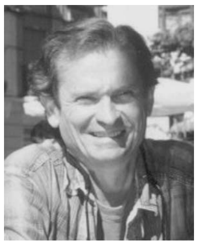
This research was supported by US National Science Foundation grant CCR-0083126, IBM, and fellowships from the US National Science Foundation and the UW Foundation. The authors are very thankful to Dieter van Melkebeek for vetting their proofs, for proofreading drafts, and for many valuable discussions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

REFERENCES

[1] D. Siewiorek and R. Swarz, *Reliable Computer Systems, Design and Evaluation,* third ed., M.A. Natick, ed. pp. 79-219, A.K. Peters, 1998.
[2] The IBM e-server pSeries 690, "Reliability, Availability, Serviceability (RAS)," technical white paper, IBM, Sept. 2001.
[3] P. Gibbons and E. Korach, "On Testing Cache-Coherent Shared Memories," *Proc. Sixth ACM Symp. Parallel Algorithms and Architectures,* pp. 177-188, 1994.
[4] P. Gibbons and E. Korach, "The Complexity of Sequential Consistency," *Proc. Fourth IEEE Symp. Parallel and Distributed Processing,* pp. 317-325, 1992.
[5] P. Gibbons and E. Korach, "Testing Shared Memories," *SIAM J. Computing,* pp. 1208-1244, Aug. 1997.
[6] P. Gibbons and E. Korach, "New Results on the Complexity of Sequential Consistency," technical report, AT&T Bell Laboratories, Murray Hill, N.J., Sept. 1993.
[7] A. Gontmakher, S. Polyakov, and A. Schuster, "Complexity of Verifying Java Shared Memory Execution," *Parallel Processing Letters,* World Scientific Publishing Company, 2003.
[8] R. Alur, K. McMillan, and D. Peled, "Model-Checking of Correctness Conditions for Concurrent Objects," *Proc. 11th Symp. Logic in Computer Science,* pp. 219-228, 1996.
[9] A. Condon and A. Hu, "Automatable Verification of Sequential Consistency," *Proc. Symp. Parallel Algorithms and Architectures,* pp. 113-121, 2001.
[10] J. Bingham, A. Condon, and A. Hu, "Toward a Decidable Notion of Sequential Consistency," *Proc. 15th ACM Symp. 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, Aug. 2003.
[12] C. Papadimitriou, *The Theory of Database Concurrency Control.* pp. 31-42, Computer Science Press Inc., 1986.
[13] R. Taylor, "Complexity of Analyzing the Synchronization Structure of Concurrent Programs," *Acta Informatica 19,* pp. 57-84, 1983.
[14] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* pp. 38-39, 95-107, 259, W.H. Freeman and Company, 1979.
[15] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. Computers,* vol. 28, no. 9, pp. 690-691, Sept. 1979.
[16] SPARC International, Inc., http://www.sparc.org, 2005.
[17] Intel Corporation, http://www.intel.com, 2005.
[18] *The PowerPC Architecture: A Specification for a New Family of RISC Processors,* second ed., C. May, E. Silha, R. Simpson, and H. Warren, eds., Morgan Kaufmann Publishers, Inc. 1994.
[19] X. Shen, Arvind, and L. Rudolph, "Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers," *Proc Int'l Symp. Computer Architecture,* 2000.
[20] K. Gharachorloo, "Memory Consistency Models for Shared-Memory Multiprocessors," WRL Research Report 95/9, 1995.
[21] P. Keleher, A. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Proc. Int'l Symp. Computer Architecture,* 1992.
[22] J. Cantin, "The Complexity of Verifying Memory Coherence," Technical Report ECE-03-01, Univ. of Wisconsin-Madison, 2003.

Jason F. Cantin received the BS degrees in electrical engineering and computer engineering from the University of Cincinnati in 2000 and, in 2002, received the MS degree in computer engineering from the University of Wisconsin. He is currently a PhD student in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. During his time as a student, he has interned six times, gaining 18 months of experience at Digital Semiconductor's Alpha Development Group, three months at Digital's Cambridge Research Lab, and eight months with IBM's e-Server Hardware Performance Group. He has filed four patent applications, has one issued patent, and has won fellowships from IBM, the US National Science Foundation, and the University of Wisconsin Foundation. He is a student member of the IEEE and the IEEE Computer Society.

Mikko H. Lipasti received the BS degree in computer engineering from Valparaiso University in 1991, and the MS (1992) and PhD (1997) degrees in electrical and computer engineering from Carnegie Mellon University. He is currently an assistant professor in the Department of Electrical and Computer Engineering at University of Wisconsin-Madison. Prior to beginning his academic career, he worked for the IBM Corporation in both software and future processor and system performance analysis and design guidance, as well as operating system kernel implementation. He recently coauthored a textbook in computer architecture, has served on numerous conference and workshop program committees, and is coorganizer of the Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD). He has published numerous conference and journal papers, in particular, the seminal work on value locality and value prediction, has filed seven patent applications, won the Best Paper Award at MICRO-29, has received IBM Invention Achievement, Patent Issuance, and Technical Recognition Awards, and was named as the Eta Kappa Nu Outstanding Young Electrical Engineer for 2002. His research interests encompass the architecture and design of high-performance desktop, server, and application-specific computer systems. He is a member of the IEEE and the IEEE Computer Society.



James E. Smith received the PhD degree in 1976 from the University of Illinois. Since then, he has been involved with a number of computer research and development projects in industry (Control Data Corporation, Astronautics Coporation, Cray Research) and as a faculty member at Wisconsin. He is a professor in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. Professor Smith and his students have conducted research in high performance processor implementations, including decoupled superscalar architectures, clustered microarchitectures, trace processors, and instruction level distributed processors. Currently, he and his group are studying the virtual machine abstraction as a technique for providing high performance and power efficiency through co-design and tight coupling of virtual machine hardware and software. He is a member of the IEEE and the IEEE Computer Society.