Verifying Sequential Consistency Using Vector Clocks

Harold W. Cain and Mikko H. Lipasti
Computer Sciences Department, Department of Electrical and Computer Engineering
The University of Wisconsin, Madison, WI 53706
cain@cs.wisc.edu mikko@ece.wisc.edu

Abstract

We present an algorithm for dynamically verifying that the execution of a multithreaded program is sequentially consistent. The algorithm uses a vector-timestamp logical time mechanism to construct and verify the acyclic nature of an execution's constraint graph.

Categories and Subject Descriptors

B.3.4 [**Memory Structures**]:Reliability, Testing, and Fault-Tolerance

General Terms

Algorithms, Theory, Verification

1 .0 Introduction

This paper addresses the problem of dynamically verifying whether the execution of a multithreaded program is sequentially consistent (SC). We describe a distributed on-line algorithm which performs this verification using a *vector-timestamp* logical time mechanism that tracks the dependencies among reads and writes to shared memory locations. The algorithm is applicable to verifying complex multiprocessor performance models, as well as performing one-pass verification of annotated multiprocessor hardware traces. However, we are also interested in applying this algorithm to the *dynamic verification* of multiprocessor systems. Dynamic verification was recently proposed for difficult-to-verify microprocessors, in which a complex microprocessor implementation is augmented with a simple checker processor[1]. Rather than verify that the complex processor is absolutely correct, a design team must only verify the checker processor. In a multiprocessor system the checker processor must also check that an execution is correct with respect to the consistency model, which is where our consistency checking algorithm is used.

The most closely related work to ours is the work of Gibbons and Korach [4], and Condon and Hu [2]. Gibbons and Korach describe a graph-based representation of a multithreaded execution called a "frontier graph", and a post-mortem algorithm that verifies the graph for SC. We present a distributed algorithm that dynamically checks the VSC-conflict problem [4]. Condon and Hu use a constraint graph representation to automate the verification of SC. Our work provides an algorithm for dynamically checking the constraint graph for correctness, in a manner more amenable to a hardware implementation than Condon and Hu's model checker.

2 .0 Preliminaries

We define the *execution* of a multithreaded program as a sequence of load and store operations performed by a set of processors P. Load (store) operations read (write) a data value v from (to) a shared memory location at address a. The execution of a memory operation can be logically split into four separate events: instantiation, initiation, global performance, and retirement. The instantiation of a memory operation occurs in program order. A memory operation is initiated when it is sent by the processor to the memory system. We use Dubois' definitions of the performance of loads and stores [3]. A store instruction s to address k is performed with respect to a processor p_i when a load instruction from k executed by p_i must return either the value generated by s or a value generated by a subsequent store to k. A store is globally performed when it is performed with respect to all processors in the system. A load is globally performed when the value it returns is fixed and cannot be altered, independent of any action of any processor, and the store which generated the value is globally performed. A store s is not globally performed until the store whose value is overwritten by s is globally performed. The last requirement, implied by Dubois et al., ensures that writes to a specific memory location are serialized [3].

A memory operation retires when it has been globally performed and when all memory operations that precede it in program order have also been globally performed. Current high performance processor implementations speculatively initiate memory operations out of program order and dynamically detect situations in which these reorderings may have violated the consistency model. If such an event occurs, the misspeculated operations are re-executed. Before a memory operation retires, it must wait until all instructions that precede the operation in program order retire. Thus while memory operations may be initiated and globally performed in an arbitrary order, they are instantiated and retired in program order.

The *local time* of a processor p_i is a monotonically increasing integer labeling each memory operation performed by p_i. In a system with n processors, we define the *global time* as an n-entry vector consisting of each processor's local time. In a shared-memory system, all communication between processors is performed by reading and writing shared memory locations. A processor p_i can perceive the local time of a processor p_k when p_i communicates with p_k through shared memory. With this communication, a processor p_i can construct a *perceived global time* vector containing p_i's local time and a stale version of every other processor's local time. The staleness of this vector with respect to a processor p_k depends on how recently that processor read or wrote the shared memory location or communicated with a third processor that read or wrote the location. The perceived global time of a processor p_i is used to infer the causal dependence of instructions subsequently executed by p_i on the operations of other processors.

An execution of a multithreaded program can be represented in terms of a directed graph called an *access graph* [5] or a *constraint graph* [2]. Nodes represent dynamic instances of load and store instructions. Edges represent the transitive ordering rela-

SPAA'02, August 10-13, 2002, Winnipeg, Manitoba, Canada.
ACM 1-58113-529-7/02/0008.

Table 1: SC Verification Algorithm Events/Actions

Event	Action(s) of Observer
Instantiation of a load or store instruction x by p_i.	1. $vinst_{ix}[i] \leftarrow vinst_{i,x-1}[i] + 1$
Global performance of a load instruction x by p_i to memory location k	1. if $(vinst_{ix}[i] < vw_k[i])$ then error 2. $\forall j \in \{1 \ldots n, j \neq i\}, vinst_{ix}[j] \leftarrow max(vinst_{i,x-1}[j], vw_k[j])$ 3. $\forall j \in \{1 \ldots n\}, vr_k[j] \leftarrow vinst_{ix}[j]$
Global performance of a store instruction x by p_i to memory location k	1. if $(vinst_{ix}[i] < vw_k[i]$ or $vinst_{ix}[i] < vr_k[i])$ then error 2. $\forall j \in \{1 \ldots n, j \neq i\}, vinst_{ix}[j] \leftarrow max(vinst_{i,x-1}[j], vw_k[j], vr_k[j])$ 3. $\forall j \in \{1 \ldots n\}, vw_k[j] \leftarrow vinst_{ix}[j]$

tions between these instructions. Using Condon and Hu's terminology, edges are classified as: *program order edges*, *inheritance edges*, *store order edges*, and *forced edges*. Inheritance edges indicate from which store operation a load inherits its value. Store order edges provide a total ordering of all store operations to the same memory location. Forced edges preserve the order between load instructions and conflicting subsequent store instructions.

3 .0 SC Verification Algorithm

If the constraint graph corresponding to an execution is acyclic, then the execution is SC, as shown by Landin et al [5]. We use a vector clock-based algorithm that dynamically maintains an execution's constraint graph and checks that graph for cycles. The vector clock representation maintains logical time in a distributed system while preserving the partial ordering of events. Vector clocks have been used extensively in the distributed systems research community, however we are not aware of any previous work verifying memory consistency models using vector clocks.

Informally, the online SC verification algorithm works as follows. For each processor in the system, there exists an observer that monitors the global performance of memory operations by that processor, meanwhile updating shared vector clock bookkeeping information. The observer also monitors the stream of instructions being instantiated and performed by the processor, and checks the perceived global time at which each memory operation is performed. If this check indicates that a processor perceived that two memory operations were performed out of order, the observer signals a violation. This algorithm assumes that processors perform memory operations in program order. We now provide a formal description of the algorithm.

3.1 Data Structures

The observer associated with each processor p_i maintains a timestamp vector $vinst_{ix}[1...n]$ of monotonically increasing, non-negative integers where n is the number of processors in the system. $vinst_{ix}[i]$ represents the local time at p_i after executing instruction x. For all $k \in \{1 \ldots n, k \neq i\}$, $vinst_{ix}[k]$ represents p_i's most recent knowledge of p_k's local time after executing instruction x. That is, $vinst_{ix}[k]$ represents the progress of processor p_k from which processor p_i most recently either received data from p_k (via a load) or overwrote (via a store) a memory location previously read or written by p_k. The entire vector $vinst_{ix}$ represents p_i's *perceived global time* after executing instruction x.

There are also two timestamp vectors associated with each individually addressable shared-memory location k: a write vector $vw_k[1...n]$, and a read vector $vr_k[1...n]$. The write (read) vector represents the perceived global time when the memory location is written (read), as perceived by the processor performing the write (read). These vectors are shared among all of the observers.

3.2 Algorithm

Initially, for each processor i, $vinst_{i0}[1 \ldots n] \leftarrow 0$. For each memory location k, $vw_k[1 \ldots n] \leftarrow 0$ and $vr_k[1 \ldots n] \leftarrow 0$.

During the execution of the system, an observer o_i associated with processor p_i monitors p_i for the instantiation and global performance of loads and stores. Upon the occurrence of each event, the observer performs the actions as specified in Table 1. As described in the table, each time a memory operation is globally performed by a processor p_i, the perceived global time at p_i is updated. Before updating the perceived global time, the observer compares the current perceived global time to the memory location's timestamp vector(s) , to confirm that the execution is consistent up to this point. If it is not, an error is signalled indicating that the current execution is inconsistent.

4 .0 Acknowledgments

This work was generously supported by NSF grants CCR-0073440, CCR-0083126, EIA-0103670 and CCR-0133437. The authors would like to thank Gordon B. Bell, Jason Cantin, Kevin Lepak, Ravi Rajwar, and Jim Smith for their insightful conversations regarding this work.

References

[1] T. Austin. "DIVA: A reliable substrate for deep submicron microarchitecture design." In *Proc. of the 32nd Intl. Symp. on Microarchitecture*, November 1999.

[2] A. Condon and A. J. Hu. "Automatable verification of sequential consistency." In *13th Symposium on Parallel Algorithms and Architectures*, January 2001.

[3] M. Dubois and C. Scheurich. "Memory access dependencies in shared-memory multiprocessors." *IEEE Transactions on Software Engineering*, 16(6):660–674, 1990.

[4] P. B. Gibbons and E. Korach. "Testing shared memories." *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[5] A. Landin, E. Hagersten, and S. Haridi. "Race-free interconnection networks and multiprocessor consistency." In *Proc. of the 18th Intl. Symp. on Comp. Architecture*, 1991.