A System-Level Specification Framework for I/O Architectures*

Mark D. Hill, Anne E. Condon, Manoj Plakal, Daniel J. Sorin
Computer Sciences Department,
University of Wisconsin - Madison,
1210 West Dayton Street, Madison, WI 53706, USA.
{markhill, condon, plakal, sorin}@cs.wisc.edu

Abstract

A computer system is useless unless it can interact with the outside world through input/output (I/O) devices. I/O systems are complex, including aspects such as memory-mapped operations, interrupts, and bus bridges. Often, I/O behavior is described for isolated devices without a formal description of how the complete I/O system behaves. The lack of an end-to-end system description makes the tasks of system programmers and hardware implementors more difficult to do correctly.

This paper proposes a framework for formally describing I/O architectures called Wisconsin I/O (WIO). WIO extends work on memory consistency models (that formally specify the behavior of normal memory) to handle considerations such as memory-mapped operations, device operations, interrupts, and operations with side effects. Specifically, WIO asks each processor or device that can issue k operation types to specify ordering requirements in a k $\times$ *k table. A system obeys WIO if there always exists a total order of all operations that respects processor and device ordering requirements and has the value of each "read" equal to the value of the most recent "write" to that address.*

This paper then presents examples of WIO specifications for systems with various memory consistency models including sequential consistency (SC), SPARC TSO, an approximation of Intel IA-32, and Compaq Alpha. Finally, we present a directory-based implementation of an SC system and a proof which shows that the implementation conforms to its WIO specification.

1 Introduction

Modern computer hardware is complex. Processors execute instructions out of program order, non-blocking caches issue coherence transactions concurrently, and system interconnects have moved well beyond simple buses that completed transactions one at a time in a total order. Fortunately, most of this complexity is hidden from software with an interface called the computer's "architecture." A computer architecture includes at least four components:

- The *instruction set architecture* gives the user-level and system-level instructions supported and how they are sequenced (usually serially at each processor).
- A *memory consistency model* (e.g., sequential consistency, SPARC Total Store Order, or Compaq Alpha) gives the behavior of memory.
- The *virtual memory architecture* specifies the structure and operation of page tables and translation buffers.
- The *Input/Output* (*I/O*) *architecture* specifies how programs interact with devices and memory.

This paper examines issues in the often-neglected I/O architecture. The I/O architecture of modern systems is complex, as illustrated by Smotherman's venerable I/O taxonomy [15]. It includes at least the following three aspects. First, software, usually operating system device drivers, must be able to direct device activity and obtain device data and status. Most systems today implement this with *memory-mapped operations*. A memory-mapped operation is a normal memory-reference instruction (e.g., load or store) whose address is translated by the virtual memory system to an uncacheable physical address that is recognized by a device instead of regular memory. A device responds to a load by replying with a data word and possibly performing an internal side-effect (e.g., popping the read data from a queue). A device responds to a store by absorbing the written data and possibly performing an internal side-effect (e.g., sending an external message). Precise device behavior is device specific. Second, most systems support *interrupts* whereby a device sends a message to a processor. A processor receiving an interrupt may ignore it or jump to an interrupt handler to process it. Interrupts may transfer no information (beyond the fact that an interrupt has occurred), include a "type" field, or possibly include one or more data fields. Third, most systems support *direct memory access* (DMA). With DMA, a device can transfer data into or out of a region of memory (e.g., 4Kbytes) without processor intervention.

An example that uses all three types of mechanisms is a disk read. A processor begins a disk read by using memory-mapped stores to inform a disk controller of the source address on disk, the destination address in memory, and the length. The processor then switches to other work, because a disk access takes millions of instruction opportunities. The disk controller obtains the data from disk and uses DMA to copy it to memory. When the DMA is complete, the disk controller interrupts the processor to inform it that the data is available.

A problem with current I/O architectures is that the behaviors of disks, network interfaces, frame buffers, I/O buses (e.g., PCI), system interconnects (e.g., PentiumPro bus and SGI Origin 2000 interconnect), and bus bridges (that connect I/O buses and system interconnects) are usually specified in isolation. This tendency to specify things in isolation makes it difficult to take a "systems" view to answer system-level questions, such as:

- What must a programmer to do (if anything) if he or she wants to ensure that two memory-mapped stores to the same device arrive in the same order?

* This technical report adds Appendix A, "Proof that an Implementation Satisfies WIO," to the paper that appears in the *Proceedings of the 11th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1999.

- How does a disk implementor ensure that a DMA is complete so that an interrupt signalling that the data is in memory does not arrive at a processor before the data is in memory?
- How much is the system interconnect or bus bridge designer allowed to reorder transactions to improve performance or reduce cost?

This paper proposes a formal framework, called *Wisconsin I/O* (WIO), that facilitates the specification of the systems aspects of an I/O architecture. WIO builds on work on memory consistency models that formally specifies the behavior of loads and stores to normal memory. Lamport's sequential consistency (SC), for example, requires that "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [10]." WIO, however, must deal with several issues not included in most memory consistency models: (a) a processor can perform more operations (e.g., memory-mapped stores and incoming interrupts), (b) devices perform operations (e.g., disks doing DMA and sending interrupts), (c) operations can have side effects (e.g., a memory-mapped load popping data or an interrupt invoking a handler), and (d) it may not be a good idea to require that the order among operations issued by the same processor/device (e.g., memory-mapped stores to different devices) always be preserved by the system.

To handle this generality, WIO asks each processor or device to provide a table of ordering requirements. If a processor/device can issue k types of operations, the required table is $k \times k$, where the i,j-th entry specifies the ordering the system should preserve from an operation of type i to an operation of type j issued later by that processor or device in program order (i.e., in the order specified by the processor or device's program). A disk, for example, might never need order to be preserved among the multiple memory operations needed to implement a DMA. A system with p processors and d devices obeys WIO if there exists a total order of all of the operations issued in the system that respects the subset of the program order of each processor and device, as specified in the $p+d$ tables given as parameters, such that the value of each "read" is equal to the value of the most recent "write" to that address.[1]

This paper is organized as follows. In Section 2, we discuss related work. Section 3 presents the model of the system we are studying. Section 4 explains the orderings that are used to specify the I/O architecture for a system whose memory model is SC, and it defines Wisconsin I/O consistency based on these orderings. Section 5 extends the framework to incorporate other memory consistency models. Section 6 describes a system with I/O that is complex enough to illustrate real issues, but simple enough to be presented in a conference paper. In Section 7, we show that the system described in Section 6 obeys Wisconsin I/O. Finally, Section 8 summarizes our results.

We see this paper as having two contributions. First, we present a formal framework for describing system aspects of I/O architectures. Second, we illustrate that framework in a complete example.

1. The same table can be re-used for homogeneous processors and devices. We precisely define "read" and "write" in later sections.

2 Related Work

The publicly available work that we found related to formally specifying the system behavior of I/O architectures is sparse. As discussed in the introduction, work on memory consistency models is related [1]. Prior to our current understanding of memory consistency models, memory behavior was sometimes specified individually by hardware elements (e.g., processor, cache, interconnect, and memory module). Memory consistency models replaced this disjoint view with a specification of how the system behaves on accesses to main memory. We seek to extend a similar approach to include accesses across I/O bridges and to devices.

Many popular architectures, such as Intel Architecture-32 (IA-32) and Sun SPARC, appear not to formally specify their I/O behavior (at least not in the public literature). An exception is Compaq Alpha, where Chapter 8 of its specification [14] discusses ordering of accesses across I/O bridges, DMA, interrupts, etc. Specifically, a processor accesses a device by posting information to a "mailbox" at an I/O bridge. The bridge then performs the access on the I/O bus. The processor can then poll the bridge to see when the operation completes or to obtain any return value. DMA is modeled with "control" accesses that are completely ordered and "data" accesses that are not ordered. Consistent with Alpha's relaxed memory consistency model, memory barriers are needed in most cases where software desires ordering (e.g., after receiving an interrupt for a DMA completion and before reading the newly-written memory buffer). We seek to define a more general I/O framework than the specific one Alpha chose and to more formally specify how I/O fits into the partial and total orders of a system's memory consistency model.

3 System Model

We consider a system consisting of multiple processor nodes, device nodes, and memory nodes that share an interconnect. Figure 1 shows two possible organizations of such a multiprocessor system, where shared memory is implemented using either a broadcast bus or a point-to-point network with directories [5]. The addressable memory space is divided into ordinary cacheable memory space and uncacheable I/O space. We now describe each part of the system.

Processor Nodes: A processor node consists of a processor, cache, network interface, and interrupt register. Each processor "issues" a stream of operations, and these operations are listed and described in Table 1. Note that LD and LDio are not necessarily different opcodes; in many machines, they are disambiguated by the address they access. We classify operations based on whether they read data (ReadOP) or write data (WriteOP). If the cache cannot satisfy an operation, it initiates a transaction (these will be described in Section 6) to either obtain the requested data in the necessary state or interact with an I/O device. The cache is also allowed to proactively issue transactions, such as prefetches. In addition, the processor (logically) checks its interrupt register, which we consider to be part of the I/O space, before executing each instruction in its program, and it may branch to an interrupt handler depending on the value of the interrupt register.

Device Nodes: We model a device node as a device processor and a device memory. Each device processor can issue operations to its device memory. In addition, it can also issue operations which lead to transactions across the I/O bridge (via the I/O bus). These

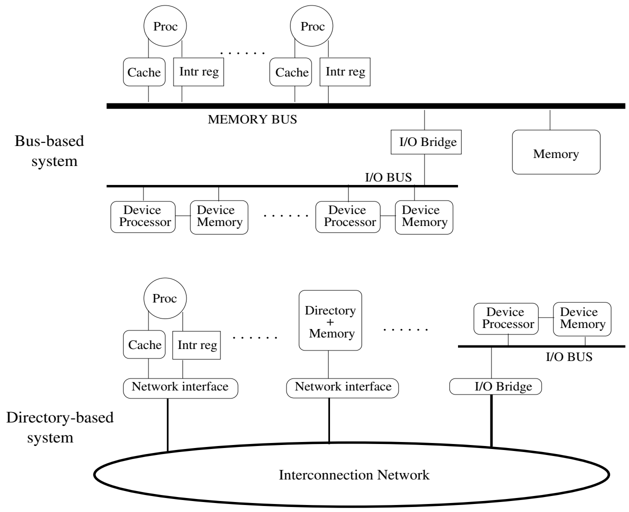

FIGURE 1. Possible System Organizations

TABLE 1. Processor Operations

Operation	Class	Description
LD	ReadOP	Load - load word from ordinary memory space
ST	WriteOP	Store - store word to ordinary memory space
LDio	ReadOP	Load I/O - load word from I/O space
STio	WriteOP	Store I/O - store word to I/O space

requests allow a device to read and write blocks of ordinary cacheable memory (via DMA) and to write to a processor node's interrupt register. The list of device operations is shown in Table 2.

A request from a processor node to a device memory can "cause" the device to "do something useful." For example, a write to a disk controller status register can trigger a disk read to begin. This is modeled by the device processor executing some sort of a program (that specifies the device behavior) which, for example, makes it sit in a loop, check for external requests to its device memory, and then do certain things (e.g., manipulate physical devices) before possibly doing an operation to its device memory or to ordinary memory. The device program will usually be hard-coded in the device controller circuits, while the requests from processor nodes will be part of a device driver that is part of the operating system. Note that, in general, the execution of a subroutine by the device in *response* to an external request to device memory needs to be made atomic with respect to other external requests to device memory. This avoids data races in accessing device memory locations.

Memory nodes: Memory nodes contain some portion of the ordinary shared memory space. In a system that uses a directory protocol, they also contain the portion of the directory associated with that memory. Memory nodes respond to requests made by processor nodes and device nodes. Their behavior is defined by the specific coherence protocol used by the system.

TABLE 2. Device Operations

Operation	Class	Description
LDio	ReadOP	Load I/O - load word from device memory (I/O space)
STio	WriteOP	Store I/O - store word to device memory (I/O space)
INT	-	Interrupt - send an interrupt to a processor node
LDblk	ReadOP	Load Block - load cache block from ordinary memory
STblk	WriteOP	Store Block - store cache block to ordinary memory

Interconnect: The interconnect consists of the network between the processor and memory nodes and the I/O bridges. This could either be a broadcast bus or a general point-to-point interconnectionnetwork. The I/O bridges are responsible for handling traffic between the processor and memory nodes, and the device nodes. Note that, while we allow a system to contain multiple bridges, we do assume that a single device is accessible via exactly one bridge. This could perhaps be extended to systems where devices are accessible through multiple bridges (for fault-tolerance reasons), by assuming that only one device-bridge pairing is active at any point in time.

Example: We now present an example that shows how this model can be used to describe a common I/O scenario. Table 3 illustrates disk reads, which, for example, might be initiated by the operating system for paging virtual memory or for accessing files in a disk-based file-system. In the example, the first operand of a memory operation is the destination and the second operand is the source. The example assumes a hypothetical disk controller with device registers DR0, DR1, DR2, and DR3 mapped into I/O address space. These registers are used to control the initial disk block number to read, the starting memory address of the buffer which will contain the data to be read, the length of the buffer, and the command (Read) to be executed. In the table, physical time flows downwards. The final STio to DR3 (the command register) immediately "triggers" the device to read all of the device registers and to set up the disk to do the read. Data is transferred using DMA between the disk and coherent memory via physical disk reads and STblks. It is useful to note here that most operating systems would make sure that these STblks do not generate any unnecessary coherence activity by invalidating all shared and modified copies (to speed up the DMA). Finally, an interrupt is generated when the disk controller has finished the DMA. This triggers the interrupt handler at the processor which can then use the data.

4 An I/O Framework for Sequential Consistency

As the example in the previous section shows, certain orderings between operations are required in order to get device operations to work. The objective of our framework is to concisely capture the orderings required of a system. In this section, we present a version of our framework for ordering the memory and I/O operations in a system where the memory model is sequential consistency (SC). Section 5 will address systems with other memory models. We begin with the ordering at individual processors and devices, and

TABLE 3. Disk Read

	Processor	Disk Controller
Setup	STio Block, [DR0]	
	STio Address, [DR1]	
	STio Length, [DR2]	
	STio Read-Cmd, [DR3]	
DMA		Read DR0, DR1, DR2, DR3 and set up disk read
		Read in data from disk, issue STblk for each cache block of data to appropriate address
		INT
	Interrupt handler runs	
Use data	LD R1, [Address]	
	ST [Address+4], R1	

then we incorporate these orderings into a framework for system-wide ordering.

4.1 Processor and Device Ordering

In a given execution of the system, at each processor or device there is a total ordering of the operations (from the list LD, ST, LDio, STio, INT, LDblk, and STblk) that can be issued by that processor or device. Call this *program order* and denote it by $<_p$.

Let *partial program order* be any relaxation of program order at a processor or a device processor. For example, let $<_{pp}$ be the partial program order that respects program order with respect to operations to the same address and also satisfies the constraints of Tables 4 and 5, where entries in these tables use the following notation:

A: OP1 $<_{pp}$ OP2 always

D: OP1 $<_{pp}$ OP2 if the addresses of OP1 and OP2 refer to the same device

-: no ordering constraint on OP1, OP2 (if not to the same address)

The entries in the tables reflect the behavior of a hypothetical system. For example, in many systems, STios to multiple devices are not guaranteed to be ordered in any particular way. Also, there is no ordering from a STio to a subsequent LD or ST, since that would require the processor to wait for an acknowledgment from the device.

It is important to realize that a programmer who wishes to enforce ordering between operations that are not guaranteed to be ordered can create an ordering through transitivity. For example, a programmer can order a processor's LD after a STio by inserting a LDio to the same device as the STio between the two operations. Since STio $<_{pp}$ LDio and LDio $<_{pp}$ LD, we have STio $<_{pp}$ LD (for this particular sequence of three operations).

4.2 System Ordering: Wisconsin I/O Consistency for SC

Using the definition of partial program order, we can now define a system ordering which we call Wisconsin I/O ordering. The definition of Wisconsin I/O (WIO) ordering takes as a parameter an n-tuple of partial program orders, such as the 2-tuple specified by Tables 4 and 5. Let $<_W$ be a total ordering of all LD, ST, LDio, STio, INT, LDblk, STblk operations of an execution of the system. Then $<_W$ satisfies Wisconsin I/O with respect to a given partial program order if:

TABLE 4. Partial Program Order at a Processor

		Operation 2			
		LD	ST	LDio	STio
Operation 1	LD	A	A	A	A
	ST	A	A	A	A
	LDio	A	A	D	D
	STio	-	-	D	D

TABLE 5. Partial Program Order at a Device Processor

		Operation 2				
		LDio	STio	INT	LDblk	STblk
Operation 1	LDio	A	A	A	A	A
	STio	A	A	A	A	A
	INT	-	-	D	-	-
	LDblk	-	-	A	-	-
	STblk	-	-	A	-	-

1. $<_W$ respects the partial program order, and

2. the value read by every ReadOP operation is the value stored by the most recent WriteOP operation to the same address in the $<_W$ order.

In Sections 6 and 7, we will describe an implementation for an SC system and outline a proof that shows it obeys this specification.

5 An I/O Framework for Other Consistency Models

To ease presentation complexity and concentrate on I/O aspects, we have thus far assumed a memory consistency model of sequential consistency. More relaxed models, such as SPARC TSO and Compaq Alpha, can also be accommodated, and we now show how this can be accomplished. We accommodate them by changing the partial program ordering at the processor, but we leave the device processor ordering unchanged. One could easily imagine providing a WIO specification where the device ordering does not match the ordering specified in Table 5, but instead matches that of the specific device(s) being modeled.

5.1 Processor and Device Ordering

As in Section 4.1, for each memory consistency model, we will present tables of ordering requirements for partial program order at processors. In the following discussion, we do not include synchronization operations, such as Read-Modify-Write (RMW). A RMW can be thought of as an atomic operation which includes a LD and then a ST. It would be ordered such that order between a

RMW and another operation, OP2, respects the union of ordering rules between OP2 and a LD and between OP2 and a ST.

5.1.1 SPARC Total Store Order (TSO)

SPARC Total Store Order (TSO) [17] is a variant of *processor consistency* [7,8] that has been implemented on Sun multiprocessors for many years. TSO relaxes SC in that STs can be ordered after LDs which follow them in program order (so long as there are no intervening memory barriers (MB) and the two operations are to different locations). Thus, TSO sometimes allows a load to get a value from a "future" store. In real implementations, this behavior is manifest when a processor's LD returns a value from its own ST that is still on its own first-in-first-out (FIFO) write buffer and has not yet seen by other processors. It should be noted that TSO supports multiple flavors of MBs, but we only concern ourselves with the strongest (i.e., an MB that enforces order between any operation before it and any operation after it).

In previous research [3], we developed a memory model called Wisconsin TSO that is equivalent to SPARC TSO, and it eliminates the oddity of getting a value from a "future" store by splitting each ST into a $ST_{private}$ and a ST_{public}. Wisconsin TSO respects program order between $ST_{private}$s and LDs, while ST_{public}s can be delayed until the next MB in program order. In addition, ST_{public}s must also stay in program order with respect to each other. The $ST_{private}$ and ST_{public} corresponding to the same ST carry the same value. A LD gets its value from either (a) the most recent $ST_{private}$ by the same processor as the LD for which the corresponding ST_{public} has not yet occurred (if any) or (b) the most recent ST_{public} otherwise. The $ST_{private}$ or ST_{public} from which the LD gets its value is considered to be the *applicable WriteOP*. Practitioners can think of a $ST_{private}$ as a store entering a processor's FIFO write buffer, case (a) as bypassing from the write buffer, ST_{public} as a store exiting the write buffer, and case (b) as obtaining a LD's value from cache or memory.

TABLE 6. TSO: Partial Program Order at a Processor

		Operation 2					
		LD	STpriv	ST_{pub}	MB	LDio	STio
Operation 1	LD	A	A	A	A	A	A
	ST_{priv}	A	A	A[a]	A	A	A
	ST_{pub}	-	-	A	A	A	A
	MB	A	A	A	A	A	A
	LDio	A	A	A	A	D	D
	STio	-	-	-	A	D	D

a. Includes the case where both operations are caused by the same Store (i.e., OP1 is the STprivate and OP2 is the STpublic for a given ST).

This definition leads to the ordering rules shown in Table 6 for partial program order at a processor, where differences from Table 4 are shaded. Note that a programmer can enforce order from a ST_{public} to a LD by inserting an MB between them.

5.1.2 An Approximation of Intel IA-32

The Intel IA-32 memory model is similar to TSO, in that it is a variant of processor consistency. We approximate the IA-32 system ordering model by combining the TSO memory model with our interpretation of the IA-32 I/O ordering rules [4]. IA-32 has two uncached (UC) operations, LDuc and STuc, that are similar to our LDio and STio I/O operations, but UC operations are more strictly ordered. All operations before a UC operation (in program order) are ordered before the UC operation, all operations after a LDuc are ordered after the LDuc, and all STs after a STuc are ordered after the STuc. In addition to the UC operations, IA-32 has two "write combining" (WC) uncached operations, LDwc and STwc. These operations are less strictly ordered than LDio/STio operations, and they are well-suited to the access ordering requirements for a video frame buffer. There is no ordering enforced between WC operations or between a WC operation and a cacheable memory operation. Also, IA-32 has several "serializing instructions" which enforce ordering in much the same way as memory barriers, and we will simply refer to them as MBs.

We have made two simplifications in this description of IA-32. First, IA-32 has several flavors of cacheable memory operations, including Write-through, Write-back, and Write-protected, but we will fold them all into LD/ST operations. Second, it supports IN and OUT I/O instructions, which are not memory-mapped I/O, but instead directly access I/O ports. These I/O instructions are ordered just as strongly as MBs, and we do not include them here.

Table 7 shows the ordering rules at a processor obeying our approximation of IA-32. Once again, differences from the SC table are shaded. Notice the extra ordering requirements of the LDuc/STuc compared to those of the LDio/STio in Table 4.

5.1.3 Compaq Alpha

The Compaq (DEC) Alpha memory model [14] is a weakly consistent model that relaxes the ordering requirements at a given processor between any accesses to different memory locations unless ordering is explicitly stated with the use of a Memory Barrier (MB). The Alpha memory model is formally defined through the use of two orders that must be observed with respect to memory accesses. The first order, program *issue order*, is a partial order on the memory operations (LDs, STs) issued by a given processor. Issue order relaxes program order in that there is no order between accesses to different locations without intervening MBs. Issue order enforces order between accesses to the same location, order between any access and an MB, and order between MBs. The second order, access order, is a total order of operations on a single memory location (regardless of the processors that issued them).

We previously defined an equivalent memory model, called Wisconsin Alpha [3], where an execution of an implementation satisfies the Wisconsin Alpha memory model if there exists a total ordering of all loads, stores, and MBs, such that:

- all of the issue orders are respected, and
- a load returns the value of the most recent store to the same location in this total order.

This definition of Wisconsin Alpha is reflected in the partial program ordering rules shown in Table 8. Notice that there are no ordering requirements between LDs and STs (unless they are to the

TABLE 7. "IA-32": Partial Program Order at a Processor

Operation 1	Operation 2							
	LD	STpriv	STpub	MB	LDuc	STuc	LDwc	STwc
LD	A	A	A	A	A	A	-	-
STpriv	A	A	A[a]	A	A	A	-	-
STpub	-	-	A	A	A	A	-	-
MB	A	A	A	A	A	A	A	A
LDuc	A	A	A	A	A	A	A	A
STuc	-	A	A	A	A	A	-	A
LDwc	-	-	-	A	A	A	-	-
STwc	-	-	-	A	A	A	-	-

a. Includes the case where both operations are for the same ST (i.e., OP1 is the STprivate and OP2 is the STpublic for a given ST).

TABLE 8. Alpha: Partial Program Order at a Processor

Operation 1	Operation 2				
	LD	ST	MB	LDio	STio
LD	-	-	A	A	A
ST	-	-	A	A	A
MB	A	A	A	A	A
LDio	A	A	A	D	D
STio	-	-	A	D	D

same address). To enforce order between them requires inserting an MB between them, which creates the order LD/ST $<_W$ MB $<_W$ LD/ST.

5.1.4 Release Consistency

Release consistency (RC), particularly the RCpc flavor, is one of the most relaxed memory consistency models [7]. To define consistency models like this, Gharachorloo et al. developed a general framework for memory consistency models, where writes are broken into p+1 sub-operations, where p is the number of processors in the system [6]. This framework, in turn, is based on a system abstraction developed by Collier [2].

Along these lines, we could expand our partial program order tables to reflect that a store in an RC system could appear to be broken up into a $ST_{private}$ and many ST_{public}s, with one ST_{public} at each processor. The applicable WriteOP for a LD would be either the $ST_{private}$ or the ST_{public} at that processor. Moreover, RC has two new operations, Acquires and Releases, which can be considered to be types of MBs for our purposes. Acquires and Releases would be included in the processor partial program order table, and the ordering required among them would depend on the flavor of RC. For example, the ordering between acquires and releases in an RCpc system would be the same as the ordering between LDs and STs in a processor consistent system (e.g., TSO). This approach, however, could lead to large, unwieldy tables.

5.2 WIO Consistency for General Memory Models

Extending the definition of WIO from Section 4.2 to incorporate memory models other than SC requires that we:

- Add any new operations, such as LDwc and STwc (which are a ReadOP and a WriteOP, respectively).
- Define what the applicable WriteOPs are for a ReadOP. For example, in TSO, the applicable WriteOP for a LD is the most recent $ST_{private}$ at that processor unless the corresponding ST_{public} is also before the LD, in which case it is the most recent ST_{public}.
- Change WIO property 2 to read:

2. the value read by every ReadOP operation is the value stored by the most recent *applicable* WriteOP operation to the same address in the $<_W$ order.

6 An Implementation that Obeys WIO for SC

So far, we have provided abstract specifications of systems that include I/O. We now provide a concrete implementation that aims to conform to the WIO specification for SC systems presented in Section 4. In this section, we specify a sequentially consistent directory-based system consisting of the components described in Section 3. This description builds upon the directory protocol described in Plakal et al. [12]. The description is divided into descriptions of the processor nodes, interconnect, I/O devices, bridge and memory nodes.

Processor nodes: The cache receives a stream of LD/ST/LDio/STio operations from the processor and, if it cannot satisfy a request, it issues a transaction.[1] The complete list of transactions, including block transfer transactions (Rblk/Wblk) that can only be issued by devices and which will be discussed later, are shown in Table 9. Cache coherence transactions (GETX/GETS/UPG/WB) are directed to the home of the memory block in question (i.e., the memory node which contains the directory information for that

1. As noted earlier, caches can also proactively issue transactions without receiving an operation from their processors.

block). I/O transactions (Rio/Wio) are directed to a specific I/O device and also contain an address of a location within the memory of the device (and, if Wio, the data to write as well). The granularity of access for an I/O transaction is one word (for simplicity of exposition). Rios generate a reply message from which the cache extracts a register value and passes it to the processor. Wios do not generate any reply messages from the target device; in the case that a processor issues a Wio and desires a response, it can subsequently query the device with a Rio. Note that *each* LDio or STio generates *exactly one* Rio or Wio (respectively). This is unlike normal cacheable memory transactions where, for example, multiple LDs or STs may be issued to the same block after a single GETX brought it into the cache.

TABLE 9. Transactions

Transaction	Description
GETX	Get Exclusive access
GETS	Get Shared access
UPG	Upgrade (Shared to Exclusive) access
WB	Write Back
Rio	Read I/O - read word from I/O space
Wio	Write I/O - write word to I/O space
Rblk	Read Block - read cache block from ordinary memory
Wblk	Write Block - write cache block to ordinary memory

Processor nodes must conform to the list of behavior requirements specified in Section 2.4 of Plakal et al. [12] (e.g., a processor node maintains at most one outstanding request for each block). They must also conform to the ordering restrictions laid out in Table 4. Thus, they do not issue a LD/ST until all LDios preceding it in program order have been "performed" (i.e., the reply has been written into the register by the cache).

A processor node's network interface sends all transactions from the cache into the interconnection network. In addition, the network interface will pass a Wio coming from the network to the processor's interrupt register. It also passes all replies to transactions to the cache.

Interconnect: The network ensures point-to-point order between a processor node and a device node, and it ensures reliable and eventual delivery of all messages.

Bridge: The I/O bridge performs the following functions: it receives Rio/Wios from processor nodes and broadcasts them on the I/O Bus (this has to be done in order of receipt on a per-device basis); sends Wio replies from device memory to processor nodes; sends Wios (to interrupt registers) from device processors to processor nodes; participates in Rblk/Wblk transactions (discussed below) and broadcasts completion acknowledgments on the I/O bus. The I/O bridge must obey certain rules. It provides sufficient buffering such that it does not have to deny (negative acknowledgment or NACK) requests sent by processors or devices. It also handles the re-try of its own NACKed requests (to memory nodes). No order is observed in the issue/overlap of Rblk/Wblk transactions.

Device Nodes: Each device processor can issue LDio/STios to its device memory and INTs to processor interrupt registers. INT operations are converted to Wio transactions by the I/O bridge. These are directed to a specific processor's interrupt register and do not generate reply messages. In addition, a device can also issue LDblk and STblk requests, and these operations are converted to Rblk and Wblk transactions by the bridge and are directed to the home node. The data payload for both requests is a processor cache line (equal to a block of memory at a memory node, which is equal to the coherence unit for the entire system). Both requests generate acknowledgments (ACKs) on the I/O bus (from the bridge) and, in the case of the Rblk, the ACK contains the data as well. A Wblk request carries the data with it. Also, each LDblk/STblk *will* generate exactly *one* Rblk/Wblk (just as with LDio/STios and Rio/Wios).

Each device memory receives a stream of LDio/STios from its device processor. In addition, it also receives a stream of Rio/Wios from the bridge (via the I/O bus) which it logically treats as LDio/STios. These two streams are interleaved arbitrarily by the device memory. For each incoming Rio, the device memory sends (via the bus and the bridge) the value of that location back to the node that sent the Rio. LDio/STios operate on device memory like a processor's LD/STs operate on its cache.

The device processor must obey the ordering rules specified in Table 5. For example, an INT is not issued until all LDblk/STblks preceding it in "device program order" have been performed (i.e., an ACK has been received from the bridge for the corresponding Rblk/Wblk).

Memory Nodes: Memory nodes operate as described in Plakal et al. [12] (with respect to directory state and transactions), with the following modifications for handling Rblk/Wblk transactions. Protocol actions depend on the state of the block at the home node for both transactions.

Rblk:

- *Idle* or *Shared*: the home sends the block to the bridge, which broadcasts an ACK with the data on the I/O bus.
- *Exclusive*: the home changes state to *Busy-Rblk*, removes the current owner's ID from CACHED, and forwards the request to the current owner. The owner sends the block to the bridge, invalidates the block in its cache, and sends an update message (with the block) to the home, which changes the state to *Idle* and writes the block to memory. The bridge receives the block and broadcasts an ACK along with the data on the I/O bus.
- *Busy-Any*: the home NACKs the request.

Wblk:

- *Idle*: the home stores the block to memory and sends an ACK to the bridge. The bridge sends an ACK to the device (via broadcast on the I/O Bus).
- *Shared*: the home stores the block to memory, sends invalidations to all shared copies, sends a count of the copies to the bridge and changes the state to *Busy-Wblk*. The bridge waits until it receives all ACKs for the invalidations before broad-

TABLE 10. Example 1

D1	P2	D3
	send Wio W1 to D1	
recv Wio W1	send Rio W1 to D1	
STio W1	send Rio W2 to D3	
recv Rio W1		recv Rio W2
LDio W1; send to P2		LDio W2; send to P2
	receive W2	
	LDio W2	
	receive W1	
	LDio W1	

TABLE 11. Example 2

D3	P4	P5
	GETX B	
	send Wio W2 to D3	
		recv INV B
	recv acks/data for B	
revc Wio W2		
STio W2	ST B	

casting the transaction completion ACK on the I/O Bus. The bridge also then sends an ack to the home which enables it to change its state to *Idle*.

- *Exclusive*: the home stores the block to memory, sends an invalidation to the (previous) owner, sends an ACK to the bridge, and changes the state to *Busy-Wblk*. The former owner invalidates its copy and sends an ack to the bridge, which then sends an ACK to the device and to the home (which then changes its state to *Idle*).
- *Busy-Any*: the home NACKs the request.

Note that we now have two new "busy" home states, *Busy-Rblk* and *Busy-Wblk*, which serve similar roles as the busy states used in the original directory protocol. These modifications make some formerly impossible situations possible. In particular, *Writeback* requests may find the home busy. One solution is to modify this transaction case:

- *Writeback* on home *Busy-Rblk* or *Busy-Wblk*: This is the same as when the home is *Busy-Shared*.

7 Proof that the Implementation Satisfies WIO

We show correctness of the implementation described in Section 6 as follows. We will use a verification technique based on Lamport's logical clocks [9] that we have successfully applied to systems without I/O [16, 12, 3]. The technique relies on being able to assign timestamps to operations in a system and then proving that the ordering induced by the timestamps has the properties required of the implementation. In order to apply our verification technique, we first describe a timestamping scheme that logically orders all ReadOps and WriteOps that occur in any given execution of the protocol. Second, we show that the resulting total order satisfies properties 1 and 2 of WIO consistency, as in Section 4.2 for SC. A detailed specification of our correctness proof can be found in Appendix A; the following is a short overview of our approach.

To specify the timestamping scheme, we augment processors, directory, and device processors (all referred to as nodes) with logical clocks. We stress that these clocks are simply conceptual tools, not part of the actual protocol. Using these clocks, a unique timestamp is assigned to each read and write. In addition, a transaction that causes a node to change its access permission to a block of data or word of I/O is timestamped by that node. Thus, a transaction may be timestamped by several nodes. Roughly, when an event (i.e. read, write, or transaction) to be timestamped "happens" at a node, the clock is moved forward in time and the updated time on the clock is assigned to that event. Of course, events are not atomic and so a central aspect of the timestamping method is the determination, from the protocol specification, of exactly when (and where) events are timestamped (and thus when they are considered to "happen"). In this way, the timestamping scheme provides a single, total ordering of all key events in the system. The correctness proof then shows that the real system behaves just as if the events happened atomically, in the order given by the timestamping scheme.

Tables 10, 11, and 12 are examples that illustrate how the timestamping scheme works and help in reasoning about correctness of our protocol. We need to describe one further aspect of timestamps before getting to our examples. Timestamps are split into three non-negative integral components: global time, local time, and processor ID. As will become clearer from the example, global timestamps help to order transactions which happen *across nodes*, whereas local timestamps help to order read and write operations that happen internal to a node. Processor ID simply acts as a tie-breaker between operations with the same global and local timestamps.

The first example, shown in Table 10, shows one processor, P2, that communicates with two devices, namely D1 and D3. P2 simply does a write followed by a read to a word W1 of D1, followed by a read to a word W2 of D3. Because the network preserves point to point ordering of messages, D1 first receives the "Wio W1" request, and then the "Rio W1" request; D1 performs these operations in order and returns the value of W1 to P2. Meanwhile, D3 handles the "Rio W2" request and returns the value of W2 to P2.

TABLE 12. Combined example with timestamps. Initially, all clocks (global.local) are set to 0.0.

D1	P2	D3	P4	P5
	send Wio W1 to D1		GETX B	
1.0.1 recv Wio W1	send Rio W1 to D1		send Wio W2 to D3	
1.1.1 STio W1	send Rio W2 to D3			1.0.5 recv INV B
2.0.1 recv Rio W1		1.0.3 recv Rio W2	2.0.4 recv acks/data for B	
2.1.1 LDio W1; send to P2		1.1.3 LDio W2; send to P2		
	receive W2		2.1.4 ST B	
	LDio W2			
	receive W1			
	LDio W1			
		2.0.3 recv Wio W2		
		2.1.3 STio W2		

Table 12 shows how these reads and writes are timestamped. In our timestamping scheme, reads and writes to device memory are timestamped at the device (thus ensuring that, in the resulting total ordering, the value of a read is that of the most recent write to the same word). The Wio and Rio requests to D1 are considered to be transactions and so D1 assigns global time 1 to the Wio and global time 2 to the Rio request. As with all transactions, the local timestamp for each of these is 0, and the final component of the timestamp is the device ID, which is 1 in our example. When the (local) "STio W1" is performed by D1, the local time is incremented, and thus the timestamp is 1.1.1. Similarly, the timestamp of the "LDio W1" operation is 2.1.1, and the events at D3 are timestamped in a manner consistent with those at D1. Thus, the "STio W1" appears before the "LDio W1" operations at D1. This is consistent with our specification in Table 4 that reads and writes to a common device (in this case, D1) by a processor should respect program order. Also note that, regardless of the relative order in real time of the "LDio W1 at D1" and "LDio W2 at D3," the "LDio W1" happens before the "LDio W2" in timestamp order simply because D1's clock is further along than D3's clock when they perform these operations. The timestamps assigned to these operations are also independent of whether P2 receives the value of W2 before or after P2 receives the value for W1. So, although the "Rio W1" appears before "Rio W2" in P2's program order, the "LDio W2" appears before the "LDio W1" in timestamp order. Again, this is consistent with Table 4, which that specifies LDios to different devices are not constrained to respect program order.

Our second example, in Table 11, concerns a processor P4 that receives exclusive permission for block B, causing processor P5 to invalidate its copy of block B. In addition, P4 sends a "Wio W2" to D3. Table 12 shows how transactions and operations at D3, P4, and P5 are timestamped. The timestamping rules specify that the global timestamp assigned by P4 to the GETX transaction must be later than the corresponding INValidate at P5. Imagine that acks sent to P4 from P5 include the timestamp of the INValidate. Also, in contrast with the fact that reads and writes to devices are timestamped at the device, reads and writes to cacheable memory (and thus the "ST B" operation at P4) are timestamped at the processor performing the operation. This is because permissions for the block reside at the processor, whereas permissions for a word of device memory always reside at the device.

Note that in Table 12, at any single node, the logical timestamps are always increasing in real time, while timestamps may be "out of order" across nodes in real time. Finally, note that the logical timestamps provide a total ordering of all reads and writes; this total ordering obtained in our example can be easily seen to satisfy the conditions of Section 4.2.

8 Conclusions

Although I/O devices are integral parts of computer systems and having clean I/O architectures would offer benefits, the commercial systems with which we are familiar tend to use ad hoc, complex, and undocumented interfaces. In this paper, we have proposed a framework called Wisconsin I/O for formally describing I/O architectures. WIO is an extension of research on memory consistency models that incorporates memory-mapped I/O, interrupts, and device operations that cause side effects. WIO is defined through ordering requirements at each processor and device, and a system is considered to obey WIO if there exists a total order of all operations that satisfies these ordering requirements such that the value of every read is equal to the value of the most recent write. We outlined how to use Lamport clocks to prove that an example system that we specified satisfies its WIO specification.

The framework presented here for specifying and analyzing systems with I/O can be generalized in several ways that were not presented earlier in order to simplify the discussion. For example, unlike in Section 6, we can model I/O bridges that do not have enough buffering to ensure that they can sink all incoming requests. Also, the definition of Wisconsin I/O consistency is parameterized by a n-tuple of partial program orders and is therefore easily generalized to handle an arbitrary set of local ordering rules. In the extreme case, each processor and each device would have its own table of partial program orders.

Acknowledgments

We would like to thank Sarita Adve, Bob Cypher, Andy Glew, Gil Neiger, and the anonymous referees for their helpful comments and suggestions. The authors, however, take responsibility for the views expressed in this paper.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, pages 66–76, December 1996.

[2] William W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, Inc., 1992.

[3] Anne E. Condon, Mark D. Hill, Manoj Plakal, and Daniel J. Sorin. Using Lamport Clocks to Reason About Relaxed Memory Models. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, January 1999.

[4] Intel Corporation. *Pentium Pro Family Developer's Manual, Version 3: Operating System Writer's Manual*. January 1996.

[5] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[6] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Specifying System Requirements for Memory Consistency Models. Technical Report CS-TR-1199, University of Wisconsin – Madison, December 1993. See also URL ftp://ftp.cs.wisc.edu/tech-reports/reports/93/tr1199.ps.Z.

[7] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[8] J. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, 1989.

[9] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[11] James P. Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, Denver, CO, June 1997.

[12] Manoj Plakal, Daniel J. Sorin, Anne E. Condon, and Mark D. Hill. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In *Proceedings of the 10th Annual ACM Symposium on Parallel Architectures and Algorithms*, Puerto Vallarta, Mexico, June 28–July 2 1998.

[13] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus for Large SMPs. *Hot Interconnects IV*, pages 41–52, 1996.

[14] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[15] Mark Smotherman. A Sequencing-Based Taxonomy of I/O Systems and Review of Historical Machines. *Computer Architecture News*, 17(5):10–15, September 1989. See also URL http://www.cs.clemson.edu/~mark/io.ps.

[16] Daniel J. Sorin, Manoj Plakal, Mark D. Hill, and Anne E. Condon. Lamport Clocks: Reasoning About Shared-Memory Correctness. Technical Report CS-TR-1367, University of Wisconsin-Madison, March 1998. See also URL ftp://ftp.cs.wisc.edu/tech-reports/reports/98/tr1367.ps.Z.

[17] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual, Version 9*. Prentice Hall, 1994. SPARC International, Inc.

Appendix A: Proof that an Implementation Satisfies WIO[1]

In this section, we will demonstrate that the implementation described in Section 6 satisfies the definition of WIO. We will use a verification technique based on Lamport's logical clocks that we have successfully applied to systems without I/O [16, 12, 3]. The technique relies on being able to assign timestamps to operations in a system and then proving that the ordering induced by the timestamps has the properties required of the implementation. Section A.1 provides background on our verification technique, Section A.2 describes the timestamping scheme for our implementation, and Section A.3 provides the proof of correctness of the implementation. Both the timestamping scheme and proof are intended to be modest extensions of those presented in our previous work [12].

A.1 Background to Lamport Clocks[2]

Our previous work on using Lamport Clocks to verify shared-memory multiprocessor systems [12,16] proved that implementations (without I/O) using a SGI Origin 2000-like [11,5] directory protocol and a Sun Gigaplane-like [13] split-transaction bus protocol both implement SC. Both implementations use three-state invalidation-based coherence protocols. We have also extended this research to use Lamport clocks to prove that systems obey two relaxed memory consistency models, SPARC TSO and Compaq Alpha [3].

Our reasoning method associates logical timestamps with loads, stores, and coherence events. We call our method *Lamport Clocks*, because our timestamping modestly extends the logical timestamps Lamport developed for distributed systems [9]. Lamport associated a counter with each host. The counter is incremented on local events and its value is used to timestamp outgoing messages. On message receipt, a host sets its counter to one greater than the maximum of its former time and the timestamp of the incoming message. Timestamp ties are broken with host ID. In this manner, Lamport creates a total order using these logical timestamps where causality flows with increasing logical time.

Our timestamping scheme extends Lamport's 2-tuple timestamps to three-tuples: **<global . local . node-id>**, where **global** takes precedence over **local,** and **local** takes precedence over **node-id** (e.g., 3.10.11 < 4.2.1). Coherence messages, or transactions, carry global timestamps. In addition, global timestamps order LD and ST operations relative to transactions. Local timestamps are assigned to LD and ST operations in order to preserve program order in Lamport time among operations that have the same global timestamp.

1. This appendix is present in the technical report version of this paper but not in the version that appears in the *Proceedings of the 11th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1999.

2. This summary is similar to the summary we present in Section 2.1 of Condon et al. [3].

They enable an unbounded number of LD/ST operations between transactions. Node-ID, the third component of a Lamport timestamp, is used as an arbitrary tiebreaker between two operations with the same global and local timestamps, thus ensuring that all LD and ST operations are totally ordered.

A.2 Timestamping Scheme for Our Implementation

Before we present the timestamping scheme, we would like to define some concepts and make some changes which will make the timestamping and the proof simpler to express and understand.

First, we split up Rblk and Wblk transactions into two steps: RBlk-Start/End and WBlk-Start/End, respectively. The reasoning behind this is as follows: cache coherence transactions (e.g., a GETX) will bring a block into a processor cache where it can be accessed until it is removed via another transaction (e.g., a WB or an incoming invalidation generated by another GETX). On the other hand, RBlk/Wblk transactions access a cache block but they do not give the device permission to do more than one operation (LDblk/STblk). It is as if the LDblk/STblk was immediately followed by a transaction that removed the device's access to the block. Conceptually breaking RBlk and WBlk into Start and End transactions unifies cache coherence and DMA transactions into one framework and simplifies the timestamping and the proof. This was not done earlier (in Section 6) to avoid confusing the reader with extra detail. The changes to the protocol are minimal: every RBlk/WBlk transaction is now regarded as a RBlk/WBlk-Start transaction. After such a transaction succeeds, a device node is now capable of performing a LDblk/STblk operation. The Rblk-End/Wblk-End is considered to conceptually occur when the transaction is complete.

Consistent with our previous work [12], we introduce the notion of a per-block *A-state* (address-state) at a node to describe the home node's view of that node's access to that block of memory. The A-state can be one of A_I (*Idle*), A_S (*Shared*), or A_X (*Exclusive*). The A-state of a block at a node changes as it participates in transactions for that node (either initiated by it or forwarded to it by the home). The A-state is set to A_I when the node receives an invalidation or a forwarded *Get-Exclusive*, or an acknowledgment for its own *Writeback* request. The A-state is set to A_S when the node receives a downgrade, or a response to its own *Get-Shared* request. Finally, the A-state is set to A_X when the node receives a response to its own *Upgrade* or *Get-Exclusive* request, along with all associated invalidation acknowledgments. In addition, we now define the A-state of a device node for a block B of memory to change to A_S or A_X when it performs a RBlk-Start or WBlk-Start, and that it change to A_I on a RBlk-End or WBlk-End. Similarly, after a RBlk/WBlk-Start transaction, the home node's A-state will change to A_I or A_S according as the final home state for that block is *Idle* or *Shared* respectively. After a RBlk/WBlk-End transaction, the home node's A-state will change to A_X if the final home state for that block (after the corresponding RBlk/WBlk-Start) was *Idle*.

We assign timestamps to the operations and transactions defined in Tables 1, 2, and 9 (with RBlk and WBlk split up as described above). The rules listed in Tables 13, 14, and 15 indicate the assignment of the global and local components of the timestamp for each kind of operation/transaction. Note that transactions do not need a local timestamp and could be assigned some arbitrary local timestamp (e.g., zero so that a transaction gets ordered before operations with the same global timestamp).

TABLE 13. Processor node timestamping

Operation/ Transaction	Global Timestamp	Local Timestamp	Node ID
LD, ST	current global clock	1 + current local clock	processor
LDio	global timestamp of corresponding Rio (sent)	1	device
STio	global timestamp of corresponding Wio (sent)	1	device
P-UP	1 + max {global clock, timestamps assigned to P-UP by all other nodes that downgrade as a result of P-UP}	0	processor
P-DOWN	1 + global clock	0	processor
Rio (sent)	*only timestamped at device*		
Wio (sent)	*only timestamped at device*		
Wio (recv)	1 + max {global clock, global timestamp of device when Wio was sent}	0[a]	processor

a. Timestamp is 0, but the clock is set to 1. This ensures that LDio/STios issued by a processor get a local timestamp of 1, while those issued by a device get a local timestamp of 2 or greater.

Conceptually, each node (processor/memory/device) maintains a global and local clock which get updated in real time for operations and transactions. To do this in a well-defined manner, we define a *timestamping order* which is a per-node total order which decides the order in which operations and timestamps get assigned timestamps. Operations enter the timestamping order of a node at the point in real time when they are retired (i.e., they cannot be undone due to mis-speculation handling), and operations are retired in a real time order that is consistent with program order. If more than one operation is committed at the same point in real time, they can be ordered arbitrarily in the timestamping order. Transactions enter the timestamping order of a node at the point in real time when the corresponding A-state change occurs at that node[1].

The timestamping rules given below also determine the maintenance of the per-processor clocks in that a node updates its global and local clocks to equal the corresponding timestamp of each

1. There is the exceptional case of *Get-Shared* transactions at the home for a *Shared* block. In this case, we consider the timestamp to be assigned at the point that the home sends the block to the requester, i.e., when the A-state "changes" from A_S to A_S.

TABLE 14. Memory node timestamping

Transaction	Global Timestamp
M-UP	1 + max {current global clock, timestamps assigned to M-UP by the nodes that downgrade as a result of M-UP}
M-DOWN	1 + current global clock
Rblk-Start	1 + max {current global clock, global timestamp of device when Rblk-Start was sent, global timestamp assigned to Rblk-Start by Exclusive node that downgrades as a result of Rblk-Start (if any)}
RBlk-End	1 + current global clock
Wblk-Start	1 + max{current global clock, global timestamp of device when Wblk-Start was sent, global timestamp assigned to Wblk-Start by all nodes that downgrade as a result of Wblk-Start (if any)}
WBlk-End	1 + current global clock

operation/transaction it timestamps in timestamping order. Any increase in the global clock value causes the local clock to be reset to zero before it is updated as specified by the rule. There are a few cases where a transaction originating at a node is timestamped elsewhere (e.g., the Wio at a device corresponding to an INT). The assignment of this timestamp causes the local node's global clock to get incremented (if necessary). For purposes of timestamping,

we consider a bridge to be part of each device node, and all transactions in which a bridge participates on behalf of a device node will update the clocks of that device node.

Processor nodes: Let P-UP be a transaction that causes an increase in coherence permissions (upgrade) at processor node p_i (GETX, GETS, or UPG by p_i), and let P-DOWN be a transaction that causes a decrease in coherence permissions (downgrade) at p_i (WB by p_i, GETX by p_j for a block that p_i has Shared or Exclusive, UPG by p_j for a block that p_i has Shared, GETS by p_j for a block that p_i has Exclusive, or Rblk/Wblk by a device for a block that p_i has Shared or Exclusive). Then the processor node timestamping rules are as shown in Table 13.

Memory nodes: Let M-UP be a transaction that causes an increase in permissions at memory node m_i (WB by p_i), and let M-DOWN be a transaction that causes a decrease in permissions at m_i (GETS, GETX, or UPG by p_i). With these definitions of M-UP and M-DOWN, the timestamping rules for memory nodes are as shown in Table 14. The memory node timestamps transactions in the real-time order in which they are processed. In the case of transactions that involve transient Busy states, the "current global clock" corresponds to the global clock at the time the Busy state is entered, while the timestamp of the transaction is assigned when the memory enters a non-transient state (*Idle*, *Shared*, *Exclusive*).

Device nodes: A device node timestamps operations and transactions as shown in Table 15.

TABLE 15. Device node timestamping

Operation/ Transaction	Global Timestamp	Local Timestamp	Node ID
LDio, STio	current global clock	1 + current local clock	device
INT	global timestamp of corresponding Wio	1	processor
LDblk	global timestamp of corresponding Rblk-Start	1	memory
STblk	global timestamp of corresponding Wblk-Start	1	memory
Rio (recv)	1 + max{global clock, global timestamp of sender when Rio was sent}	0[a]	device
Wio (recv)	1 + max {global clock, global timestamp of sender when Wio was sent}.	0[a]	device
Wio (sent)	*only timestamped at processor*		
Rblk-Start	*only timestamped at memory*		
RBlk-End	*only timestamped at memory*		
Wblk-Start	*only timestamped at memory*		
WBlk-End	*only timestamped at memory*		

a. See footnote under Table 13.

A.3 Proof of Correctness of Our Implementation

To prove WIO, it is sufficient to show that there is a total order of operations such that the orderings in Tables 4 and 5 are respected and such that every Read-OP gets the value of the most recent Write-OP. The timestamping scheme ensures the total order and, combined with the protocol specification, ensures that Tables 4 and 5 are respected. LDios and STios to device memory are ordered at the device in the order in which they are performed, so a LDio must get the value of the most recent STio. Now we will prove that every LD/LDblk gets the value of the most recent ST/STblk.

The proof that we provide here is very similar in structure to the proof that we provided in our previous work [12]. In what follows, we first outline how definitions from our previous work can be extended to the implementation presented in this paper. We then summarize the claims and lemmas that are used in the main theorem. The changes in the statements of these results (relative to our previous work in SPAA'98 [12]) are in underlined bold.

The consistency model is established using the concept of *coherence epochs*. An epoch is an interval of logical time during which a node has read-only or read-write access to a block of data. In the rest of the paper, we assume a *block* to be a fixed-size, contiguous, aligned section of memory (usually equal to the cache line size). Also, LDs and STs operate on *words*, where we assume that a word is contained in a block and is aligned at a word boundary. Our scheme could be extended to handle LDs and STs on sub-units of a word (half-words or bytes) which need not be aligned. However, this makes the specification of the memory models very tedious without any gain in insight or clarity.

Transactions on a given block are serialized by the block's directory. Hence, we can speak about a sequence of transactions on the same block where the ordering is implied by their serialization at the directory. For each node N, a sequence of t transactions on block B (where the order among transactions is seen at the Home) defines a unique sequence $A_{(1)}$, $A_{(2)}$,..., $A_{(t)}$ of associated A-states for N, given some initial A-state value at N. If $A_{(i)}$ is not equal to $A_{(i-1)}$ for some $i \geq 1$, we say that the i^{th} transaction in the sequence "affects" N and that the transaction "implies that N's A-state for block B change from $A_{(i-1)}$ to $A_{(i)}$". For example, consider a single block of memory and three nodes: N_1 (processor), N_2 (device) and N_3 (memory). Suppose that both N_1 and N_2 start out with an initial A-state of A_I and N_3 starts with A_X. Let the sequence of transactions at N_3 be N_1's *Get-Exclusive*, N_2's *RBlk-Start* and N_2's *RBlk-End*. Then the sequence of A-states for N_1, N_2 and N_3 are A_I, A_X, A_I, A_I; A_I, A_I, A_X, A_I and A_X, A_I, A_I, A_X respectively. The *Get-Exclusive* affects N_1 and N_3, while the *RBlk-Start/End* affect N_2 and N_3. In the special case that a node is the directory, we say that it is also affected by all transactions resulting from *Get-Shared* requests, even though no change in the A-state at the directory may be implied by such a transaction.

Each transaction implies an "upgrade" of A-state (i.e. change from state A_I to A_S, from A_I to A_X, or from A_S to A_X) at exactly one node. For example, a RDblk-Start causes an upgrade at the device, a downgrade at memory, and possibly a downgrade at a processor. Also, each transaction implies a "downgrade" of A-state (i.e. change from A_X to A_S, from A_X to A_I, or from A_S to A_I) at zero or more nodes. In the special case that node N is the directory, we say that N's A-state "downgrades" as a result of every *Get-Shared* transaction, even though its A-state may not be changed by the transaction. On each transaction, exactly one node upgrades and zero or more nodes downgrade.

The definitions of "affects" and "implies" in the previous two paragraphs depend only on the sequence of transactions on block B at B's directory. In Claim 2 below, we show that the protocol specification ensures that, at every node, the actual sequence of changes to the A-state for block B occurs in the order implied by the serialization of the transactions at B's directory, even though messages on successive transactions may be received out of order by a node.

Claim 1: For each transaction T, a message is sent to every processor affected by T. Also, if **processor** N upgrades as a result of T, exactly those nodes that are affected by transaction T (other than N) send a message to N.

Claim 2: The sequence of A-state changes on block B at a node occurs in real time in the order implied by the serialization of transactions on block B at its directory.

Claim 3: For a transaction T on block B,

(a) The timestamps of the downgrades associated with T are less than or equal to the timestamp of the upgrade associated with T.

(b) The timestamp of the upgrade associated with T is less than the timestamp of the upgrade associated with any transaction T' on block B occurring after T in the serialization order at the directory, so long as one of T or T' is a Get-Exclusive or Writeback **or WBlk-Start**.

Claim 4: Every LD/ST **or LDblk/STblk** operation on block B at processor p_i is bound[1] to the most recent (in Lamport time at p_i) transaction on block B that affects p_i.

By construction, the Lamport ordering of LDs and STs within any processor is consistent with program order. Therefore, to prove sequential consistency, it is sufficient to show that the value of every load equals the value of the most recent store.

Recall that a coherence epoch is simply a Lamport time interval [t1,t2) during which a node has access to a block. All operations that have global timestamp t where $t_1 \leq t < t_2$ are contained in epoch $[t_1,t_2)$. A shared or exclusive epoch for block B at node N starts at time t_1 if a transaction with timestamp t_1 (at N) implies that N's A-state for block B changes to A_S or A_X respectively. The epoch ends at time t_2, where t_2 is N's timestamp of the next transaction on B that implies a change in A-state at N.

Lemma 1 shows that two processors cannot have "conflicting" permission to the same block at the same (Lamport) time. Lemma 2 states that processors do operations within appropriate epochs. Finally, Lemma 3 shows that the "correct" block value is passed among processors and the directory between epochs.

Lemma 1: Exclusive epochs for block B do not overlap with other exclusive or shared epochs for block B in Lamport time.

Lemma 2: (a) Every LD/ST, **LDblk/STblk** operation on block B at p_i is contained in some epoch for block B at p_i and is bound to the transaction that caused that epoch to start. (b) Furthermore, every ST **or STblk** operation on block B at p_i is contained in some exclusive epoch for block B at p_i and is bound to the transaction that caused that epoch to start.

Lemma 3: If block B is received by node N at the start of epoch $[t_1,t_2)$, then each word w of block B equals the most recent ST **or STblk** to word w prior to t_1 or the initial value in the directory, if there is no store to word w prior to global time t_1.

The proof of the Main Theorem shows how WIO follows from the lemmas.

Main Theorem: The value of every LD **or LDblk** equals the value of the most recent ST **or STblk** or the initial value, if there has been no prior store.

1. In our previous work [12], we had defined the notion of LDs/STs being *bound* to the transaction that brought the corresponding block into the cache. Similarly, LDblk/STblk operations are bound to their corresponding RBlk/WBlk-Start transactions.