MANAGING POLYATOMIC COHERENCE AND RACES WITH REPLICATED SHARED MEMORY

H. G. Dietz and T. I. Mattox

Purdue University, School of Electrical and Computer Engineering
West Lafayette, IN 47907-1285
{hankd, tmattox}@ecn.purdue.edu
http://garage.ecn.purdue.edu/~papers/

Although many parallel computing systems allow processors to communicate with each other by loading and storing objects in an apparently shared memory, there is little agreement on the precise semantics of shared memory object access. However, if compatibility with the semantics of common high-level languages is the primary motivation for favoring a shared memory model, then it is critical that the behavior of the shared memory mechanism be consistent with this programming model.

Common shared memory implementations place all shared objects within a single physically shared memory system (e.g., shared bus), scatter shared objects across processor-local memories to implement a "distributed shared memory" (DSM) managed either by software or by cache coherence hardware, or "reflect" each memory write to the local memory of all processors. Instead, we propose a "replicated shared memory" (RSM) model which uses local memory much like reflective memory systems, but employs a more sophisticated update mechanism that yields the desired semantics.

1. Introduction

Despite the fact that machines supporting some flavor of shared memory parallelism are relatively common (e.g., products from Sequent, SGI, Cray and numerous SMP PCs), and many shared memory programming models have been proposed, there is no standard definition of shared memory. For the purpose of this paper, we suggest that shared memory is characterized by four fundamental properties:

1. The ability to transmit data between processors by use of an apparently conventional high-level language assignment statement storing into a variable that has the "shared" attribute or storage class.
2. If one or more processors load the value of a shared variable, the value loaded should be the same on all processors, and should be equal to the value most recently stored into that variable by any processor. This implies that races are resolved in a way that *is consistent with* a sequential store order. A discussion of how many systems approach this issue is [1].
3. No matter what size objects are placed in shared memory, all object accesses occur atomically without interference from adjacent object accesses. We call a system with this attribute "polyatomic."
4. Shared memory accesses can be made asynchronously by each processor independent of the others.

Of these, most traditional hardware and software methods for implementing shared memory can achieve 1 and 4. However, properties 2 and 3 remain elusive. Property 2 refers to the basic concept of coherence, which is often redefined in looser terms to permit more efficient implementation. Arbitrary-size polyatomic access, property 3, is generally viewed as incompatible with the fixed-width data paths used by hardware, and is often sacrificed with the silent assumption that only word-sized values are important (e.g., see [1]).

In contrast, the scheme presented in this paper makes no compromises on 1, 2, and 3, but sacrifices some degree of the asynchrony described in 4. While each processor is free to initiate a read from or write into shared memory at any time, a write generally will not complete until all processors have synchronized. This different approach enables very simple hardware to implement a model that is fully consistent with traditional programming language semantics, yet yields good performance.

The prototype hardware support for this semi-synchronous shared memory is TTL_PAPERS [5], the TTL implementation of Purdue's Adapter for Parallel Execution and Rapid Synchronization. This inexpensive hardware does not directly implement shared memory access, but was designed to provide very low latency barrier synchronization and aggregate communication operations to user-level UNIX processes running on a cluster of personal computers or workstations. It differs from the ideal hardware support mechanism primarily in the way it is physically connected to the processors; prototype performance is relatively low, but is sufficient both to demonstrate the approach and to accurately predict performance of an "IDEAL" bitwise aggregate network [5].

2. The Shared Memory Model

Before describing the implementation of the shared memory system, it is useful to define the details of the shared memory model and constraints on how the system can be used.

2.1. Data Format Constraints

From the perspective of high-level language code, "shared" is not an attribute of memory per se, but of specific data objects. For example, declaring `shared<int> i;` creates a (32-bit) signed integer variable (data object) whose value is to be maintained as a coherent copy on each of the processors.

Clearly, if all machines within a cluster use compatible processors, all of these copies are literally identical memory

images. However, if heterogeneous computers are used to form a cluster, each machine may use a different native representation for objects of a specific type. Continuing the above example, a 32-bit signed integer object may have a different native byte order for each different type of machine within a cluster (e.g., "big endian" versus "little endian"). Thus, to make a native-format copy of a shared object's value, the shared memory system must not only transmit the new value whenever the object is written, but also perform a data type dependent translation to the native format of each machine. For example, the VMIC "reflective memory network" [9] implements endian conversion in hardware.

In theory, incorporating type-dependent object format translation is relatively simple. Unfortunately, typed objects in most high-level languages have programmer-visible memory format dependencies that make the appropriate translation difficult or impossible to create. Fortran `COMMON` and `EQUIV` declarations permit different typed objects to access the same memory image (i.e., different typed objects can overlap each other in memory). Still more complex, the C language allows different-typed objects to overlay each other in memory using either `union` types or type coercion (e.g., using a `char` pointer to access bytes within an `int`). These type-aliasing constructs would expose the fact that different machines used different native layouts.

Thus, unless shared objects are restricted to be free of type-aliases, the copy-based shared memory model is effectively restricted to use on clusters in which **all processors use identical native data formats**.

2.2. Object Atomicity

Most shared memory programming models define all data object accesses to be atomic; any datum from a character to a double-precision floating-point number, or even an entire user-defined structure (in languages like C), is able to be loaded or stored with the assurance that the entire object is treated as a single entity. Unfortunately, it is very difficult to create a hardware/software system that efficiently provides this property. Because hardware generally transfers data in fixed-length units, there are really two separate types of atomicity violation to be considered.

The first case typically occurs when the datum being accessed is smaller than the unit of transfer; this case is often referred to as the "false sharing" problem. For example, operations on adjacent characters within a single word should be able to proceed in parallel. Thus, given a shared character array `c[2]` initialized to `"cc"`, executing `c[0]='a'` on one processor while another executes `c[1]='b'` should always result in `"ab"`. Most shared memory systems implement character store as a sequence of word fetch, byte insert, and word store, which can produce `"ac"` or `"cb"`, violating character atomicity.

The second type of atomicity failure typically occurs when the datum is larger than the unit of transfer; this problem is equally important, but often ignored. We will call this a "fragmentation" problem, since the incorrect results are caused by the handling of fragments of the object as separate entities. For example, if two processors attempt to store different 64-bit double-precision floating point values into the same variable, but memory access is accomplished atomically only on 32-bit words, it is possible that the stores could interleave such that the 64-bit result has the first 32 bits from one value and the last 32 bits from the other.

Ironically, in some systems, it is even possible for a single object reference to cause both types of atomicity failure. For example, consider storing a 32-bit value into a memory address that is not aligned on a 32-bit word boundary. Although such misaligned access is simply disallowed on many machines, processors like the Intel Pentium allow these accesses and implement them using a pair of word operations. Thus, only a fraction of each memory word is modified by the store, potentially yielding a false sharing atomicity failure. However, at the same time, the fact that two memory word accesses are required can cause a fragmentation atomicity failure. The same dual problem can occur when spanning page boundaries in a page-oriented shared memory system.

Neither of the above two types of atomicity failures is acceptable, thus, the model used in this paper ensures that **every shared object access is atomic**. In fact, shared memory references that would cause atomicity problems for other systems perform very well with RSM because they tend to reduce the overhead of transmitting the object addresses (TTL_PAPERS RSM uses software caching of addresses). Reference atomicity is completely unaffected by object size; an atomic object can be as small as one byte or as large as the entire virtual address space.

2.3. Addressing

Because shared objects can partially overlap, and shared objects can be pointers, it is very difficult to manage shared objects without causing each object to have the same logical memory address in all machines. Nearly all modern computers use address page table hardware to map the physical addresses used by a program into a standard logical address map. It is sufficient to ensure that each shared object has the same position in the logical address map for each machine.

If the exact same executable image is executed on all processors, all statically allocated data will naturally have identical logical addresses for all processors. Notice, however, that even details such as order of code modules being linked to create the executable image can result in static data being given a different layout. Further, even if all processors perform identical function calls declaring identical local variables, it is common that small differences in the execution environment (e.g., UNIX environment variable values) result in potentially different addresses for stack-allocated objects. Thus, local stack-allocated variables cannot be shared.

A more general alternative is to use a UNIX system call to dynamically allocate objects at specific logical (virtual) memory locations. Using `mmap()` or `shmat()`, a chunk of memory can be allocated at the same logical addresses across the cluster despite differences between the code images on different processors, i.e., fully-general MIMD as opposed to SPMD.

2.4. Asynchronous Update

Arbitrary asynchronous access should be allowed for all shared variables; it is this aspect of the idealized shared memory system that we will compromise for improved efficiency.

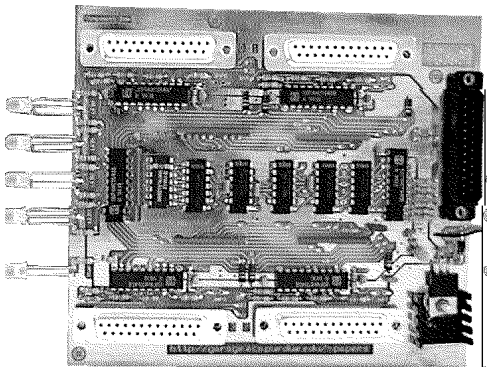
In most applications, the expected number of read accesses is greater than the expected number of write accesses. Thus, we prefer to make reads more efficient at the expense of writes — which is precisely the opposite of the cost relationship obtained from most DSM systems. The most efficient read access results from making the effect of every write immediately visible to all processors (write-thru). This makes every read a simple local access without any communication overhead and, because local actions are fully asynchronous, reads are truly asynchronous.

The remaining problem is how to implement the equivalent of a write-thru broadcast on every store into a shared object, yet obtain reasonable efficiency. Conventional workstation network hardware is not efficient with the very small message sizes implied by object write-thru; these networks are designed to optimize bandwidth, not latency. For example, the combination of Orca and Amoeba using Ethernet to implement reliable broadcast yields a peak performance of about 1,250 microseconds per broadcast [8]. Reflective memory systems, such as [9] and [6], use custom hardware optimized for low-latency broadcasts. For example, [9] quotes a memory reflect time of just 2.1 microseconds for four machines; however, the hardware is expensive, larger systems are at least linearly slower, and the transfers are word-oriented (i.e., not polyatomic).

In contrast, our approach uses hardware that is essentially a group of synchronously-sampled NAND trees across the processors. The NAND trees efficiently support low-latency broadcast by having the sending processor output the complement of its datum while all other processors output all 1 bits. However, these NAND trees also provide very fast barrier synchronization and a powerful mechanism for voting on shared memory updates so that the amount of data to be transmitted can be minimized. A four-bit-wide version of these NAND trees, and supporting circuitry, is the TTL_PAPERS hardware [5] which we will target for the detailed discussion in this paper. Using TTL_PAPERS, a typical optimized shared object write can complete in about *20 microseconds*, making *all* processors aware of the new value.

But how can "synchronously-sampled NAND trees" implement an *asynchronous* broadcast? The solution is to use the TTL_PAPERS hardware's asynchronous signaling facility (a "eureka" [4]) to trigger the synchronous cooperation of all processors to effect any pending shared memory writes. Hardware interrupt overhead is too high in comparison to communication latency, thus, processors not writing to shared objects **must occasionally poll** to see if other processors are signaling that they have a write-thru broadcast pending. Any asynchronously initiated attempt to write into a shared object blocks until all processors have also either made write requests or polled.

With a simple C++ library, reading or writing a shared object automatically polls. However, if a processor will not be accessing any shared objects while other processors might be writing shared objects, it may be necessary to manually insert explicit calls to the polling code. The worst-case delay can be bounded by setting a UNIX timer to run a polling signal handler... but this may damage the atomicity of the system if one is not careful to ensure that the UNIX signal doesn't interrupt a shared memory write already in progress.



3. TTL_PAPERS RSM Support

RSM has been implemented for clusters of PCs running Linux and communicating using the TTL_PAPERS aggregate function network [5]. The TTL_PAPERS hardware implements very low latency barrier synchronization, 4-bit wide data communication, and asynchronous parallel signaling. The barrier synchronization logic consists of two NAND trees and a flip-flop, as does the asynchronous parallel signaling logic. The 4-bit wide communication mechanism is simply four separate NAND trees. The result is very simple hardware that scales as well as NAND trees. The four-processor 960801 version, which also serves as a module to connect up to thousands of processors in a 4-ary tree, is shown in the above photo.

At this writing, we have built eleven different types of PAPERS units, and there are improvements in each generation. Space limitations do not permit us to include circuit diagrams, etc., but full hardware and software documentation for the public domain designs is available online at:

`http://garage.ecn.purdue.edu/~papers/`

TTL_PAPERS connects to each processor (each PC or workstation) using a standard parallel port (SPP) connection, which is accessed directly from the user process using either memory-mapping or I/O instructions to input from or output to hardware port registers. Although port register accesses do not suffer OS call overhead, each access takes between one and two microseconds, which is far longer than the combined software, cable, and communication logic delays. Thus, counting port register accesses gives a good measure of the cost of each operation.

It is also easy to predict the execution time on idealized hardware (henceforth called "IDEAL") by simply counting the number of 32-bit wide references that would be needed for a TTL_PAPERS-like structure connected directly to the I/O pins of a processor. Counting only the delay within the aggregate function unit (primarily a NAND gate), each operation takes about 25ns for a four-processor system, and $25*\log_4(n)$ for n

processors. Thus, an IDEAL operation might complete faster than a local memory DRAM access... which makes sense because IDEAL's logic is simpler than that of a DRAM.

3.1. `p_bcastPutz/Getz(addr, size)`

These two routines are used to perform a synchronous broadcast of an arbitrary size block of memory. The single sending processor calls `p_bcastPutz(addr, size)` and all receiving processors must call `p_bcastGetz(addr, size)` with the same `size` value. Although the TTL_PAPERS data path is only 4 bits wide, these routines use a history-based compression technique that typically requires only one 4-bit transmission for every 8 bits of data. Each such 4-bit transmission with TTL_PAPERS is currently implemented by 5 port register accesses. There are also broadcast put and get routines for transmitting a single object of any basic data type (as defined by the Gnu C Compiler), and these routines are actually slightly more efficient; for example, 32-bit integer objects are transmitted using a type-dependent compression scheme that typically requires only three 4-bit transmissions rather than the four required using the generic compressed block broadcast scheme. The result is that a typical compressed 32-bit integer broadcast can take as little as 15 to 30 microseconds. Using IDEAL, a 32-bit broadcast would take about two operations.

3.2. `s_update(addr, size)`

The `s_update(addr, size)` function is essentially the asynchronous equivalent to the synchronous `p_bcastPutz(addr, size)` and `p_bcastGetz(addr, size)`. A processor attempting to write a data object to shared memory calls this function with the obvious arguments: the virtual base address of the data object and the object size in bytes. Thus, the purpose of this call is to make the copies of the object in other processor memories consistent with the local copy which has already been updated.

The first action taken is to signal that one or more shared memory requests are pending, which is done by setting a signal and waiting for all processors to arrive at a barrier synchronization confirming that they have seen the signal. This takes as few as two port register accesses.

After the barrier synchronization, all processors determine which processors have actual shared data writes to be performed. The simplest description of the set of writers is a bit mask in which bit *i* is a 1 *iff* processor *i* has data to write. The collection of this mask using the TTL_PAPERS hardware requires no more than five port register accesses for every four processors in the cluster... or one operation for every 32 processors using IDEAL. Each processor simply outputs all 1 bits, except for the bit that it uses to indicate whether it had a datum to write. Since all processors obtain the same bit mask from the NAND logic, all now agree on which processors need to write, and the ordering of the actual write operations can be statically scheduled (e.g., using "round robin" ordering) without additional communication overhead.

As each writer is selected, it broadcasts an encoded form of the object address, size, and new data to all the other processors. For TTL_PAPERS, the SPP connection and 4-bit data path make compression worthwhile; predictions about the properties of typical writes and experiments with various application-specific encoding methods were used to determine the best compression scheme. However, the most important compression technique used is the *detection of race conditions*. When a processor that has not yet sent its store observes an overlapping store being broadcast, it appropriately adjusts its store to treat the earlier store as the winner of the race. Thus, if *k* processors are attempting to store values in the exact same object, *only the first will broadcast*; the other processors will simply withdraw their requests, allowing the first sender to win the race.

If only one processor has a write pending and we assume only 8:4 compression can be achieved on the data broadcast, typical cost for the complete TTL_PAPERS asynchronous broadcast store can be as low as 14+5*ceil(*N*/4)+5**k* port register accesses, where *N* is the number of processors and *k* is the number of data bytes in the object. For a 32-bit object and eight processors, this is 44 port register accesses or between 44 and 88 microseconds. Using eight (very slow) 386DX 33 MHz PCs running Linux, the typical time was measured to be about 60 microseconds. Put another way, this is over 16,000 RSM writes per second, or more than 20 times as fast as the mechanism used in [8]. Of course, adding more processors or sending more or more random (less compressible) data both slightly extend the broadcast time. Similarly, having more than one processor requesting a write simultaneously reduces the time per write, very significantly if races are present.

The IDEAL hardware could take as few as three operations to transmit data. Of course, the write-ordering and race-detection voting procedure should be implemented in IDEAL's hardware rather than in software.

3.3. `s_poll()`

OS context switch, and even basic hardware interrupt latency, is much longer than the typical cost of a TTL_PAPERS operation. Thus, only polling offers appropriately low latency for responding to asynchronous write broadcast requests. The TTL_PAPERS hardware must be polled to check for a parallel signal indicating that one or more shared write requests are pending. Polling costs just one port register access for TTL_PAPERS: typically one or two microseconds. It would also take only one IDEAL operation.

3.4. `s_wait()`

The `s_wait()` operation is an asynchronous request for a simple barrier synchronization. However, it is not sufficient to just perform a barrier synchronization using the hardware of TTL_PAPERS (or of IDEAL) because an asynchronously triggered barrier synchronization also implies that all asynchronous operations submitted by any processor complete before the barrier executes. This is done by having processors at a barrier poll for pending asynchronous events (updates) until all processors have arrived at the barrier. If no shared writes are pending, performance measured on the eight 386DX33 PC TTL_PAPERS cluster averaged about 10 microseconds. For IDEAL, this could be a single operation.

4. Basic RSM Compiler Technology

Our goal is to directly support polyatomic shared object access. Only the programmer and compiler know the type and size of each shared object being referenced, so one or the other must mark each shared object access appropriately. It is critical that each *access* be labeled; the same *address* may be associated both with a C `struct` and with the first member of that `struct` — nested objects requiring different atomicity. Thus, object address marking schemes like those used for DSM page tables [2][7] or reflective memory [6][9] are not sufficient.

For basic object references, it is more convenient that the compiler tag each access rather than requiring the programmer to do so. Section 4.1 describes how an ordinary C++ compiler can implement this tagging. Higher-level synchronization operations are more often specified by the programmer; Section 4.2 describes the appropriate interface.

4.1. Shared Object Templates

The "trick" that allows us to use an ordinary C++ compiler, rather than writing a custom compiler like Orca [8], is the C++ template mechanism. Our C++ shared object templates store each data object as a protected, volatile, object of the actual (base) type specified. A write operation to a shared object is implemented by the template's definition of the assignment operator specified. Perhaps less obvious is the fact that all read operations can also cause arbitrary code to be executed by simply defining a type cast operation that converts a `shared <`*actualT*`>` object into an *actualT* value. The basic (simplified) template structure is:

```
template<class actualT>
class shared {
public:
 actualT operator=(actualT rhs);
 actualT operator=(shared<actualT> rhs);
 operator actualT();
protected:
 volatile actualT data_;
}; /* shared */
```

Notice that the shared object has *exactly the same memory image* as an object of the actual type; there is no additional structure in memory. This is vital to our model, since the specification of overlapping shared memory objects (e.g., a shared `union`) should result in the same overlap that equivalent non-shared objects would have evidenced.

Clearly, correct execution may require every shared object write to cause an asynchronous shared object write request. The problem with intercepting only shared object writes is that a process which is simply looping until the local value of a shared object changes would never respond to the shared object write signal sent by another processor attempting to alter that value. Thus, the loop would be endless, resulting in a strange type of deadlock.

The solution is, of course, to ensure that `s_poll()` is called at least occasionally by all processors, but precisely when should polling occur? The easiest deadlock-avoiding solution is to simply cause each shared object read to first poll for pending shared write request signals. This is done by:

```
template<class actualT> inline
shared<actualT>::operator actualT() {
 /* poll for pending write requests */
 s_poll();
 return data_;
} /* actualT */
```

A write into a shared object is also easily implemented. However, the following code is complicated by a simple optimization that is used only for objects with actual types that define a comparison for equality operation (e.g., not `struct` nor `union`). If a write to a shared object does not change the value of the object, i.e., the value written matches the current value, there is no need for a communication. In this case, the code simply polls as it would have for a read.

```
template<class actualT> actualT
shared<actualT>::operator= (actualT rhs) {
 register actualT temp_data_ = rhs;
 if (temp_data_ != data_) {
  /* write locally and update all */
  data_ = temp_data;
  s_update(((void*)&data_),sizeof(data_));
 } else {
  /* poll for pending write requests */
  s_poll();
 }
 return data_;
}
```

Given the above, any basic or derived type can have a shared equivalent, and the C++ compiler's type tracking will cause the correct code to be generated for each reference. For example:

```
anytype a, b; shared<anytype> c;
a = b;  /* local */
a = c;  /* polls */
c = a;  /* updates (if needed...) */
```

The fundamental point about the above constructs is that shared objects *have identical access semantics* to ordinary objects. Many users would claim that this is the whole point of having a shared memory system for parallel processing.

4.2. Synchronization Functions

Because this RSM system is built using barrier synchronization hardware, it is not surprising that the programmer has access to a very efficient barrier synchronization mechanism, `s_wait()`, which can be called directly in the user C++ code. In addition, we provide a simple shared memory lock mechanism.

In most shared memory systems, the implementation of locks and other types of semaphores depends on the use of special atomic operations. However, the RSM mechanism which we propose actually provides a stronger type of atomicity for all references without any additional overhead.

A shared memory lock is declared as a variable of type `s_lock`. Each such lock is actually a shared memory object whose value is an integer between 0 and *N* for an *N*-processor system. An open lock is represented by the value

S_INITLOCK (literally, *0*). A lock held by processor *i* has the value *i+1*. The three basic operations on locks are defined in the following subsections.

4.2.1. s_initLock(lock)

To initialize a lock, the lock should simply be set to S_INITLOCK on all processors. This can be done either by statically initializing the value or by calling s_initLock(lock). The call simply sets the local value to S_INITLOCK and then uses s_update(lock, sizeof(*lock)) to update the copies in all processors.

4.2.2. s_acquire(lock)

This function is used to wait for and acquire the specified lock. The procedure is remarkably simple:

- While the local copy of the lock variable is not S_INITLOCK, call s_poll() to give the processor that currently holds the lock a chance to release it. Notice that there are no port register accesses required to test the local lock value, and only one access is required for each polling operation.
- Since the lock is now open, attempt to claim it by writing the processor number plus one into the lock and calling s_update(((void *) lock), sizeof(*lock)). Because other processors can only change the values of shared objects by performing s_update() operations in which all processors participate, and these operations are performed atomically by selecting a winner for each race, the result of this step must always be that the winning processor's value appears in all copies of the lock.
- If the local copy of the lock now holds this processor's value, this processor has been granted the lock and can return from s_acquire(). If it instead holds the value from a different processor, the lock acquisition process continues with step 1.

The reason that this simple procedure works for our shared memory model is that, unlike other types of shared memory, it is impossible for the value of a lock to change without this processor calling s_poll() or s_update().

4.2.3. s_release(lock)

To release a lock, the processor which currently holds the lock must set all copies of the lock object to S_INITLOCK. In other words, s_release(lock) is equivalent to s_initLock(lock). The only possibly desirable difference would be to prefix the operation by checking that the processor attempting to unlock currently holds that lock.

5. Conclusion

DSM systems like Ivy [7] and Treadmarks [2] use page table hardware to invoke DSM communications, with a variety of consistency models and message protocols. Using ATM network hardware and low-level AAL3/4 protocols, Treadmarks can satisfy a page miss access in 2,792 microseconds [2]. In contrast, Locust [3] integrates scheduling with an active message mechanism, yielding a DSM "fetch" time of 830 microseconds for a 4-byte object using 10 Mbits/s Ethernet. Yet, even with these slow numbers, the result was shared memory semantics that require great care in data layout and significantly unconventional access semantics.

Reflective memory systems like [6][9] offer a more natural semantics with little concern for data layout, and shared write speed of a few microseconds, through the use of optimized hardware. However, current designs do not offer as strong a consistency model as presented in this paper, are strictly word-oriented (i.e., not polyatomic), and add latency quickly as larger systems are constructed. Further, they offer no potential to optimize *across* simultaneous write operations... *n*-way write races yield the worst case performance.

In contrast, the RSM (replicated shared memory) model presented here provides the precisely the semantics desired, including arbitrary polyatomic references. It also trivially optimizes across simultaneous write operations, resolving *k*-way races in near constant time. Using just the SPP-connected TTL_PAPERS, the RSM model easily achieves typical shared object write execution times of well under 100 microseconds. This performance is achieved *without* customized compiler technology, literally using a standard C++ compiler with a special template library. Incorporating more sophisticated compiler technology, typical shared write times under 25 microseconds would be commonly observed. Using an IDEAL implementation, typical write times could rival those of local DRAM memory.

References

1. S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *IEEE Computer*, December 1996, pp. 66-76.
2. C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenpoel, "Treadmarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, February 1996, pp. 18-28.
3. T. Chiueh and M. Verma, "A Compiler-Directed Distributed Shared Memory System," Supercomputing, December 1995.
4. Cray T3D System Architecture Overview, Publication HR-04033, Cray Research, Inc., 2360 Pilot Knob Road, Mendota Heights, MN 55120, 1993.
5. R. Hoare, H. Dietz, T. Mattox, and S. Kim, "Bitwise Aggregate Networks," Proceedings of the *Eighth IEEE Symposium on Parallel and Distributed Processing*, October 1996, pp. 306-313.
6. *PCI Reflective Memory* literature, Encore Computer Corporation, Fort Lauderdale, FL, 1997. (http://www.encore.com/)
7. K. Li, "IVY: A Shared Virtual Memory System For Parallel Computing," *1988 International Conference on Parallel Processing*, August 1988, Vol. 2, pp. 94-101.
8. A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal, "Parallel Programming using Shared Objects and Broadcasting," *IEEE Computer*, August 1992, Vol. 25, No. 8, pp. 10-19.
9. *Reflective Memory White Paper*, VME Microsystems International Corporation, Huntsville, AL, Feb. 1996. (http://www.vmic.com/)