Themis: Enforcing Titanium Consistency on the NOW

Carleton Miyamoto and Ben Liblit
CS262 Semester Project Report
Computer Science Division
University of California at Berkeley
{miyamoto,liblit}@CS.Berkeley.EDU

Abstract

Titanium is a single program, multiple data (SPMD) programming language based on Java. Titanium defines a data consistency model equivalent to the Java specification. This consistency model calls for locally sequential consistency, with global consistency attained at synchronization points and arbitrary write reordering. The Titanium language is targeted at homogeneous multiprocessors on distributed memory architectures. Accesses to distant memory locations may result in communication over a network. The Themis system builds a Titanium-compliant consistency interface for Active Messages running on a NOW. This paper examines several methods for designing a Themis system that performs well for Titanium applications. Microbenchmark tests and a Conjugate Gradient Method test show the viability of each of these designs. One of the methods, a caching mechanism called OrderCache, displays promise for reducing communication loads and exploiting the locality of data within applications.

1 Introduction and Background

Titanium is a single program, multiple data (SPMD) programming language based on Java. The language is type safe and allows compilers to analyze and optimize code much more aggressively than under C style languages. Titanium was designed mainly for scientific programming on parallel, distributed memory architectures. In order to maintain compatibility with Java, Titanium inherits many of Java's features, among including the Java consistency model.

A consistency model describes the parts of a program that a compiler or runtime system may reorder to optimize performance or for convenience. Weak consistency models, such as the Java model, allow great flexibility in reordering. Although this allows the systems to better optimize the communication and computation patterns [14], the programmer has a more difficult job in understanding the language. Stronger consistency models, though simpler to understand, may severely limit the optimizations that can take place, hampering performance [3]. A balance between the extremes should be reached.

1.1 The Java Consistency Model

The Java consistency model is defined in the original Java language specification [2]. The model is described in terms of actions that may be executed by different parts of the system and the ordering that must be imposed on these actions.

This paper interprets the model to be:

1. **Locally sequentially consistent.** For a single processor, all reads and writes to a given memory location must appear to occur in exactly the order specified. For example, in the program:

```
x = 0;
x = 1;
y = x;
```

   The processor must see the final value of `y` as 1. If reads and writes could occur out of order within a processor, `y` could take on the value 0 or 1.

2. **Globally consistent at synchronization events.** At a global synchronization event, such as a barrier, all processors must agree on the values of all the variables. At a non-global synchronization event, such as entry into a critical section, the processor must see all previous updates made using that synchronization event. For example, in the program (initial value of `x` is 0):

```
critical section {
      x = x + 1;
}
```

If two processors enter this section, the first processor entering must see `x` as 0. The other must see `x` as 1.

More important than what the model demands is what the model leaves open to the implementor. For example, it allows writes between different processors to be reordered in any convenient way. For example, if two processors were simultaneously executing:

Initially, `a` is 1 and `b` is 2.	
Processor 1:	Processor 2:
`a = 3;` `b = 4;`	`Print("(" + a + "," + b + ")");`

Valid outputs for processor 2 include `(1,2)`, `(3,2)`, `(1,4)`, and `(3,4)`. This ambiguity clears the way for aggressive communication optimizations.

1.2 Active Messages on the NOW

One of the parallel architectures Titanium is targeting is the NOW, running Active Messages II (AM-II). This system has proven itself at handling and solving complex scientific problems. This combination, however, poses a few challenges to implementing an efficient system. For example, although AM-II is a reliable network delivery service, it does not make any guarantees about ordering [15]. Out-of-order packets must be handled by the higher level protocols. Doing so, however, may be tricky and require additional memory management or side stepping. This happens because of some of the assumptions (discussed later in this paper) made by AM-II. Nevertheless, AM-II is an important platform, and an efficient binding is necessary for the success of Titanium.

1.3 This Paper

This paper describes a system called Themis[1], which complies with the Java consistency model and which can be used as a communications layer between the Titanium runtime and Active Messages. In Section 2, several approaches to Themis's design are described. In Section 3, some performance numbers are provided on common kernel test cases. Sections 4 and 5 discuss future enhancements and related work. Section 6 presents conclusions about what types of systems can most benefit Titanium.

2 Design Approaches

We have implemented two independent runtime systems to enforce the Titanium consistency model on an unordered Active Messages network. The two systems take rather different approaches to the problem: one treats consistency as an issue of cache coherence, while the other treats it as a form of network packet delivery control. However, both schemes ultimately present similar interfaces for use by the SPMD application. We will first describe this common programming model, and then delve into the implementation details of the two schemes.

2.1 The Themis Programming Model

The NOW implements a distributed memory system. Each node in the network has direct access only to its own local memory, while memory on remote machines must be accessed by way of some application-deployed network communications scheme. Active Messages provides a high-performance messaging layer, but does not designate any specific policy for using the network to share data among parallel computations.
Themis builds upon the Active Messages network to provide illusion of a globally shared address space across all processors participating in a SPMD computation. Each address in this space consists of an ordered pair (processor, local-address), identifying a processing node and a local memory address on that node. Themis provides interfaces

[1] In ancient Greek mythology, Themis was a Titan, one of the divine forces of nature who preceded the rise of the gods of Olympus. Themis's domain was that of universal law, the right and natural order of all things.

to read or write the data at any global address. Each access may atomically manipulate 8, 16, or 32 bits at one time.[2] Both implementations provide the following standard entry points:

- **Synchronous writes:** store `value` into address `addr` on processor `proc`. Do not return to caller until write has completed.
- **Asynchronous writes:** initiate store of `value` into address `addr` on processor `proc`. Return to caller immediately.
- **Synchronous reads:** fetch data at address `addr` on processor `proc`. Delay until read completes, and then return fetched value to caller.
- **Asynchronous reads:** increment `signal`. Initiate fetch of data at address `addr` on processor `proc`. Return to caller immediately. When read subsequently completes, store fetched value into local address `result` and decrement `signal`.

The following additional methods manipulate read completion signals:

- **Initialization:** set the value of `signal` to zero.
- **Spin waiting:** dispatch incoming Active Message requests and replies until the value of `signal` is once again zero.

Lastly, barriers provide global synchronization:

- **Barrier:** start draining the network. Process incoming requests and replies, but do not initiate any new communication. Wait until all processors have entered the barrier and all communications have been completed, and then return to the caller.

Asynchronous reads are sufficiently complex to warrant closer attention. Since the call returns immediately, the result of the read cannot be passed to the caller as a return value. Instead, the caller designates where the result should be placed once it becomes available. For example, this might be the address of a suitably typed local variable. The caller also designates a completion signal. Completion signals are allow applications to determine when their asynchronous reads have completed. They are abstracted counters, and serve a similar role to semaphores. A signal is initialized to zero, and is incremented each time it is used in association with an asynchronous read. When a read completes, the corresponding signal is decremented. An application may issue several reads with a single signal; when the signal value has returned to zero, all of the issued reads will have completed. Figure 1 shows an example of code that uses signals and asynchronous reads to retrieve a pair of remote values.

```
static uint proc1, proc2;
static int32 *addr1, addr2;
int32 value1, value2;

/* set up the completion signal */
Signal signal;
SignalInit( &signal );

/* issue siumultaneous read requests */
Read32Async( proc1, addr1, &value1, &signal );
Read32Async( proc1, addr1, &value1, &signal );

/* wait for both reads to complete, in either order */
SignalWait( &signal );

/* values are now available for use */
        int32 sum = value1 + value2;
```

Figure 1: Using signals to coordinate reads.

[2] Although Titanium contains 64-bit “long” and “double” data types, the consistency model does not require these to be operated upon atomically. These would be manipulated in Themis using a pair of 32-bit accesses.

2.2 *OrderCache*

Within systems, there often exists a mismatch between the speed at which a device can be accessed and the rate at which accesses occur. Caching is often employed to reduce this mismatch by placing the results of past requests in a faster medium for quick reuse. A similar situation occurs between computer networks and main memory. As parallel programs continually send network requests, the network can quickly become congested and appear like a slow device. OrderCache deploys a cache on these network requests to cut the network traffic and allow fast, convenient access to often used data within main memory. OrderCache is also responsible for guaranteeing the consistency model required by the Titanium language specification.

2.2.1 Memory Layout

After considering a few different formats, OrderCache was structured as a write-back, direct mapped cache. In memory, the cache appears as in Figure 2:

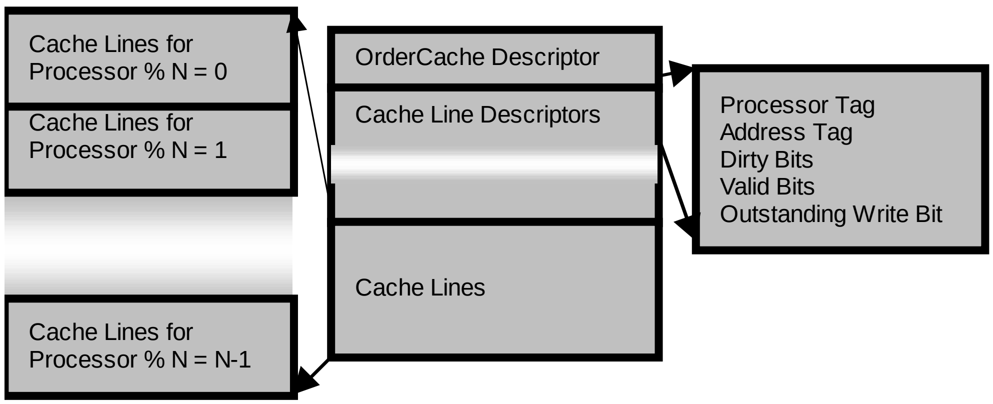

Figure 2: OrderCache memory layout.

This set of layout and cache policies was chosen for several reasons:

1. It is simple.
2. Allows for a number of optimizations. By placing the cache lines adjacent in memory away from the cache descriptors, the cache can do bulk transfers of data on adjacent cache entries. Much more care would be necessary in a more complex design.
3. No hardware support for parallel entry lookup. In hardware, set-associative style caches provide better performance by minimizing cache misses that are due to conflicts. Such caches take advantage of parallel lookups to accomplish this with no performance degradation. In software caches, however, multiple lookups cannot easily be parallelized.
4. Lookup is fast. Only a few sequences of shifts and bitwise operations are required.
5. Takes advantage of data locality. This is a well-known problem by both programmers and compilers. If a hash function were used to generate the cache indices, optimizing data placement for the cache would be much more difficult.

2.2.2 Writes

OrderCache sends two types of messages over the network between processors. Write messages write data to a processor/memory-address pair. Both synchronous and asynchronous versions exist. Writes are issued on a cache miss on a dirty page or on a synchronous write request. Upon receipt of a write request, the processor updates the value at the specified address then returns an acknowledgement of the update. Data sizes of 8-, 16-, and 32-bits are supported.

2.2.3 Reads

Reads convey data between two processors. Just as with writes, both synchronous and asynchronous versions are available. Read requests are sent on cache misses. When a read request is received, the memory is immediately read and the value returned in the acknowledgement. Data sizes of 8-, 16-, and 32-bits are supported. If an asynchronous read is used, it is possible for the cache line to become dirty before the read result propagates back to the original requestor. Two ways of dealing with this are:

1. Flush the dirty cache line and put the read result in the cache
2. Do nothing; just drop the return result

In order to keep the handlers simple and fast, the second option was chosen.

2.2.4 The Active Messages Interface

Each read or write request is packaged as an active message. In order to support single- and double-byte updates, a mask is sent with each request to determine which parts of each message are valid or which the requestor cares about. The necessary actions are performed in the request handlers and the results returned in a reply message. For a read, this message includes the result; write replies are simply acknowledgements. The reply handlers update the cache descriptors and lines as necessary.

2.2.5 Keeping Consistent

Another job of the OrderCache is to maintain the Titanium consistency model. It does this by maintaining an "outstanding write" bit within each cache descriptor. Because a global write schedule does not exist, network reordering does not pose a problem for writes originating from different processors. Within a processor, however, writes must look sequential. As a result, if multiple write requests are issued for the same memory location, a later write could overtake an earlier one and arrive first. The result is an old value placed in memory. To solve this, a bit is set whenever OrderCache sends a write message over the network but has not yet received the acknowledgement from that write. If the bit is still set and the OrderCache needs to issue another write for the same cache line, it instead enters a spin lock until the bit is cleared by the return of the first write's acknowledgement. This policy insures that, per processor, only a single outstanding write exists per memory location. OrderCache thereby avoids the network reordering problem. Finally, in order to preserve global consistency, the cache is flushed at synchronization events, such as barriers.

By using the hardware cache model, OrderCache attempts to improve performance in network systems by reducing latency and congestion. In fact, OrderCache behaves like an emulation of a weakly consistent processor cache on a shared memory architecture. Its simplicity and better direct program control, however, allow an efficient implementation in software.

Using these techniques, the OrderCache exactly implements the Java/Titanium consistency model.

2.2.6 Cache Sizes

When the OrderCache is created, two values are specified that determine the working cache size. These are:

1. number of low order bits of the address to use in the index
2. number of low order bits of the processor number to use in the index

These parameters allow an application to tune the cache structure to its specific requirements. As in a hardware cache, the high order bits of the processor number and address are stored as tags in the cache descriptor. The low order bits from these two form the index. The index is calculated so that the cache lines are equally split into processor regions (see Figure 2).

So, the high order bits of the index come from the processor number, while the low order bits come from the address. This separation into regions helps to avoid cache conflicts due to communication with multiple processors.

2.2.7 OrderNoCache

In order to help measure how caching affects consistency policies in OrderCache, a modified version, OrderNoCache, was created that eliminates caching but retains the same mechanisms for consistency. Read requests are always sent over the network, and, similarly, write requests are flushed immediately. The “outstanding write” bit is set, and a stall still occurs if the write would have mapped to a cache line where the bit was set.

2.3 Sequencer

Sequencer takes a different approach to enforcing Titanium consistency. Because the NOW is a distributed memory machine, remote memory operations necessarily become network communications. The ordering of memory operations is equivalent to the ordering of network message delivery. The challenge in enforcing the Titanium consistency model stems from the fact that Active Messages does not guarantee ordered delivery. All messages will be delivered exactly once, but they may be reordered arbitrarily in transit [15]. If messages can be reordered, then memory operations can be reordered, and the consistency model breaks down. In order to enforce the consistency model, then, we must enforce ordered processing of network messages.

Layering an ordered network upon an unordered one is a well-known challenge. For example, solving this problem was part of building TCP on top of IP [10]. In that case, the solution was to embed a sequence number within each packet. The addressee may receive packets in any order, but the embedded sequence numbers serve as reassembly instructions: packets that arrive too early are buffered on the receiving end until their logical predecessors have also been received. The Themis sequencer uses the same approach. Messages represent requests to read or write the recipient’s remote memory. Each message carries a sequence number, allowing the recipient to reassemble a properly ordered set of requests before processing them.

2.3.1 Conceptual Design: An Illustrative Example

The operation of the Themis sequencer is best demonstrated by example. Figure 3 illustrates the progress of a pair of memory accesses through the system. In the illustration, “client” and “server” denote a pair of processors, where the client wishes to access remote memory hosted on the server. “Outbound” and “inbound” denote a pair of counters, one on each processor, initially set to 1. The server also maintains a dispatch queue, described further below.

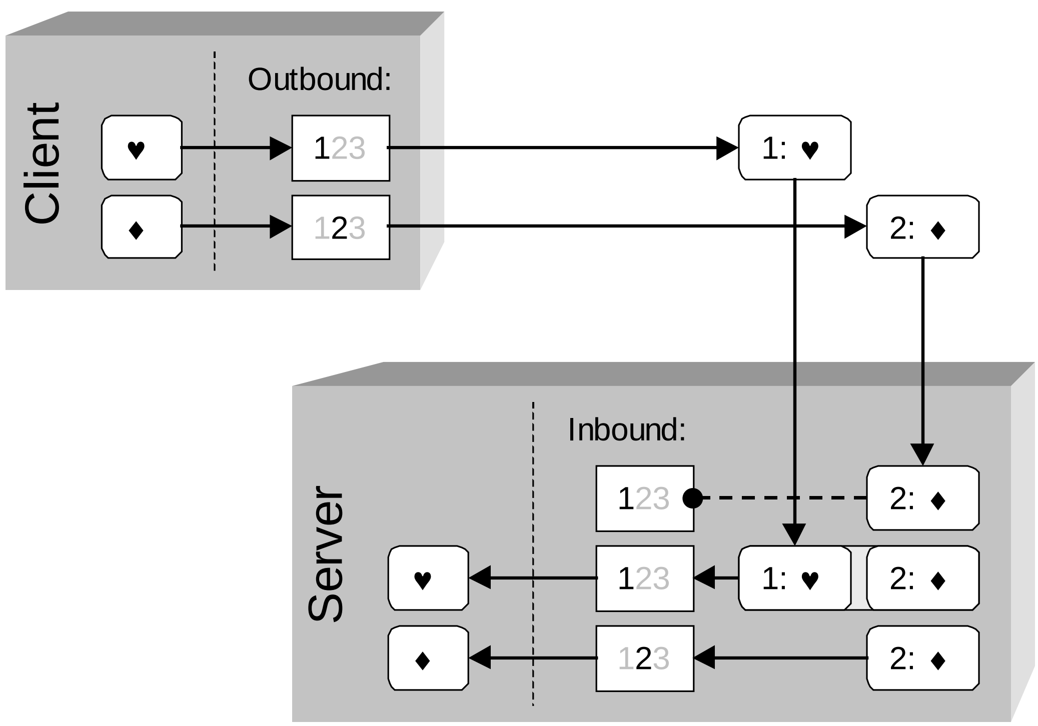

Figure 3: Sequencer overview.

The client initiates the activity by accessing a global address that physically resides on the server. This first access is denoted symbolically as “♥”. This may be either a read or a write; Sequencer’s overall behavior is the same in either case. Simply consider that “♥” represents the unique parameters of the memory operation, such as a global address and a value in the case of a write. The “♥” access is passed into the client sequencer library, which augments the request with the current outbound sequence number, 1. The assembled message, “1: ♥”, is injected into the unordered Active Message network for delivery to the server. Without waiting for a reply, the client now issues a second request, denoted “♦”. The request is augmented with the current outbound sequence number, which is now 2. The assembled message, “2: ♦”, is also injected into the unordered Active Message network for delivery to the server.

Whenever there are multiple outstanding messages, the possibility exists that the messages may be reordered. Suppose that the client’s second request, “2: ♦”, is received first by the server. The request is inserted into the server’s dispatch queue, which is ordered by sequence number. The server now examines the head of the queue, and will compare the leading item’s sequence number, 2, with its own inbound counter, 1. Since the two do not match, the “2: ♦” request must have arrived out of order. In this sense, the inbound counter represents the next numbered request for which the server is waiting. The out-of-order request remains in the dispatch queue, to be fulfilled later.

Eventually, the server will receive the client’s first message, “1: ♥”. This request is inserted into the server’s dispatch queue; its lower sequence number forces it to the front of the queue, ahead of “2: ♦”. Again, the server examines the head of its queue, and will compare the leading request’s sequence number, 1, with its own inbound counter, 1. Since the two match, the head of the queue contains the next sequential request from the client. The leading item, “1: ♥”, is dequeued. The server’s inbound counter is incremented, to 2. The requested read or write operation, “♥”, is applied. In the case of a read, the value at the supplied local address is fetched and sent back to the client. In the case of a write, the supplied value is assigned into the supplied local address, and the client notified that the change is complete.

The server now reexamines the head of its dispatch queue. The leading item, “2: ♦”, has a sequence number that matches the server’s inbound counter. The request is dequeued, the “♦” operation is completed, and the inbound counter is incremented to 3. Should the client eventually issue a third request, the server will be ready and waiting for it to arrive.

2.3.2 Generalized Operation

In general, every processor performs reads and writes, and every processor hosts a portion of the global address space. Thus, every processor must be able to act as both a client and server relative to every other processor. Any ordered pair of processors may perform as client and server for a given memory operation. Therefore, Sequencer maintains a *vector* of outbound and inbound counters on each processor. For a parallel computation across p processors, each processor maintains p outbound counters and p inbound counters. Each of the p outbound counters records the next sequence number to be used for future requests sent to one other processor. Each of the p inbound counters records the next expected sequence number for future requests received from one other processor.

Thus far, we have concentrated on how requests are ordered and dispatched by servers. Eventually, clients must be notified that their memory operations have completed. Writes require nothing more than a simple confirmation; reads require that the retrieved value be returned to the requesting client. Since these replies are sent using Active Messages, they are just as vulnerable to reordering. When a client receives a reply from a server, it must be able to determine which specific request has just completed. A third vector of p counters on each processor accomplishes this feat. When a server finally dispatches a given request, the sequence number of that request is embedded in the reply, along with the server’s own processor number. Thus, the count of outbound messages is complemented by a count of replies to those messages. Each time a reply is received, the client increments the “acknowledgement” counter corresponding to the server issuing the reply.

The acknowledgement counters allow clients to make guarantees about which requests must have completed on their corresponding servers. When an acknowledgement counter reaches some value n, the client must have received replies to exactly n of its outstanding requests to that server. This implies that the corresponding server must have received, ordered, and dispatched at least the first n of the client’s requests. (The server might have processed more than n, since some further replies may still be propagating across the network.) If the client is waiting for a specific request to finish, it is sufficient to wait until the acknowledgement counter for the server in question has reached the sequence number embedded in the original request. Similarly, if all of a client’s outbound counters exactly match their corresponding acknowledgement counters, than all outstanding requests have been completed and all replies have been received. This property is used to “drain” the network at a global barrier.

2.3.3 Active Messages Implementation

Each processor sends a single Active Message request whenever it wishes to read or write another processor’s memory. In both cases, the message identifies the requesting processor and the client sequence number associated with the request.

Conceptually, servers do not reply to client requests until those requests have been ordered and dispatched. However, Active Messages mandates that each request message handler must issue exactly one reply before returning. Since we cannot control the order in which request handlers are called, a server may find that it is required to reply to a message that it knows to have arrived out of order. In such cases, a special empty reply message is issued that effectively notifies the client that its request has been received but postponed. Clients take no action in response to these messages, but sending them satisfies the Active Messages requirement for one immediate reply to each request. Later on, when the server is able to order and dispatch the request, it will initiate a new communication to the client notifying it that the original request has finally completed. Since this is a new communication, Active Messages now demands that the client issue a reply. As before, an empty message is sent and is simply ignored by the server.

Each server’s dispatch queue is implemented as a singly linked list, anchored at the head. When a new request arrives, it is not placed into the queue immediately. Rather, its sequence number is first checked against the appropriate inbound counter. If the request has arrived without being reordered, it is dispatched and replied to immediately, without ever being enqueued. Otherwise, a new queue node is allocated, and the request postponed. The queue must grow dynamically, because servers have no control over how many out-of-order messages they may need to buffer at once. A free list of recyclable queue nodes helps to avoid excessive allocation within message handlers. In theory, the use of a single queue for all inbound requests creates the possibility of head-of-line blocking. In practice, reordering is rare enough that this does not have a significant performance impact.

2.3.4 Sequencer Consistency Model

The Sequencer, as OrderCache, can also supply sufficient guarantees to support the Java model. Note, however, that the Sequencer actually provides a slightly stronger model, though this fact is not used in the tests. It can guarantee a global write ordering. This means that although writes can be reordered, all processors will see that same order.

In the Java model, not only can writes occur in any order, but because global consistency does not occur until a synchronization event, one processor can see a two writes occurring in a different order than another processor. For example, if three processors were simultaneously executing the following:

Initially, `x` is 0 and `y` is 0.		
Processor 1:	Processor 2:	Processor 3:
`x = 1;` `y = 2;`	`print(" (" + x + "," + y + ")");`	`print("(" + x + "," + y + ")");`

Processor 2 could output `(1,0)`, while processor 3 could output `(0,2)`. The sequencer, however, will guarantee that if processor 2 outputted `(1,0)`, that processor 3 will output only `(0,0)`, `(1,0)`, or `(1,2)`.

3 Performance Analyses

3.1 Experimental Setup

The OrderCache, OrderNoCache, and Sequencer have been implemented as user-space code libraries, each in roughly 1,400 lines of commented C code. All three schemes use the GLUnix 1.0a network operating system [8] on top of Solaris 2.5.1 to coordinate SMP execution across a network of workstations. Active Messages II release 3.1 provides the message exchange primitives. Both libraries and benchmarks were compiled using gcc 2.7.2.2 with aggressive optimization (“`-O3`”) enabled.

All benchmarks were performed on the Berkeley NOW. The NOW is a network of up to one hundred Sun UltraSPARC Model 170 workstations, running at 167 MHz, with 128 megabytes of main memory and 512 kilobytes of secondary cache. Each workstation has a single Myricom M2F network interface card on the SBUS, containing

128 kilobytes of SRAM card memory and a 37.5 MHz LANai processor. The machines are interconnected with ten eight-port Myrinet M2F switches, with throughput of 160 megabytes per second per port.

3.2 Microbenchmarks

We have devised a suite of microbenchmarks with which to study the behavior of OrderCache and Sequencer. These tests compare the performance of each consistency scheme under a variety of memory access patterns. Tests were conducted on the Berkeley NOW, while varying such parameters as size of access, size of cache, and number of participating processors.

At the start of each test, each participating processor establishes a 1024 element array of values for other processors to access. The elements of the array may be 8-, 16-, or 32-bit values, depending upon the particular test permutation. These arrays are placed at known locations in each processor's memory, allowing them to be referenced easily by other processors. Thus, the matrix of $p \times 1024$ values defines the globally shared address space upon which all accesses take place.

During a benchmark run, each processor performs $2 \times p \times 1024$ global memory accesses. Thus, in a 16-processor test each processor would access 32,768 global memory locations. Each individual access is randomly chosen to be either a single synchronous read or a single asynchronous write, with reads biased to be three times more common than writes. Three global access patterns were used:

Sequential access. In the sequential access test, each processor sequentially accesses every member of every array on every processor, including itself. The entire global address space is traversed twice in this manner.

Random access. In the random access test, each processor randomly selects a processor and an array offset, and accesses that point in the global address space. Random selection is repeated $2 \times p \times 1024$ times.

Wander access. In the wander access test, each processor initially selects a random processor and array offset. Starting from that point in the global address space, the processor begins a Cartesian randomized walk. On each access, the processor either accesses the same location (20% chance), an adjacent array value on the same processor (60% chance), or the same array value on an adjacent processor (20% chance).

3.2.1 Aggregate Performance

Figure 4 shows an overview of the performance of our three candidate consistency enforcement schemes. Recall that OrderNoCache is a variant of OrderCache with certain key features disabled. OrderNoCache provides no read caching, so that every read requires a network communication. OrderNoCache also provides no write buffering. Instead, dirtied cache entries are flushed immediately.

OrderCache operates twice as fast as Sequencer overall. The most significant speedups may be seen in the wander access test. This is consistent with the fact that wander access produces the best data locality. When locality is high, OrderCache is able to service many memory requests locally, without requiring network communication. The Sequencer, on the other hand, operates identically regardless of locality. Thus, we observe that its performance is consistent across all access patterns.

Although OrderCache benefits from good locality, it can also suffer from false sharing. When OrderCache is flushing a dirty cache line, no subsequent flushes may occur on the same line until the first one completes. The OrderCache must stall successive flushes in order to preserve the write ordering requirements of the consistency model. If writes are common, the risk of such a flush stall increases. This accounts for the degraded performance of OrderNoCache relative to Sequencer. Both schemes submit writes to the network immediately. However, Sequencer never needs to wait for a write to complete, whereas OrderNoCache may incur a significant number of flush stalls.

Several other factors contribute to OrderCache's impressive performance. On 8- and 16-bit accesses, OrderCache actually manipulates 32 bits of data at a time. For small reads, this leads to prefetching of adjacent bytes, improving the odds of subsequent cache read hits when locality is good. For small writes, four neighboring dirty bytes can be flushed using a single network communication. Sequencer is unable to take advantage of this sort of byte clustering; we have found that it presents consistent performance not only across access patterns but across access bit sizes as well.

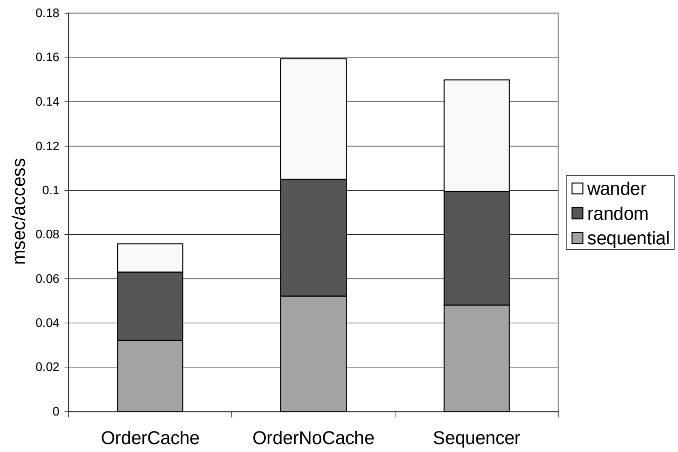

Figure 4: Aggregate performance of Themis consistency schemes.

3.2.2 Reordering Rates

A critical component of design of Sequencer is that it has a fast path for dispatching messages that arrive in order. Properly ordered messages never enter the queue, and complete processing requires only two messages. Reordered messages must be enqueued first, and complete processing requires four messages. When messages arrive in proper order, Sequencer should be able to perform as well as a raw Active Messages network. If reordering predominates, though, Sequencer's performance would rapidly degrade. In practice, we have found that reordering is extremely rare on the current incarnation of the Berkeley NOW. The complete Sequencer benchmark suite issued 250,675,086 Active Message requests from clients to servers. Only 114 requests arrived on servers out of order. Reordering only took place on the sixteen-processor benchmarks, and no processor's dispatch queue ever held more than four items at one time.

3.2.3 Barrier performance

The aggregate performance data presented earlier demonstrates that OrderCache is able to translate good cache hit rates into greatly improved performance. Because OrderCache is a software cache, it can be made arbitrarily large; one might assume that enormous caches would lead to enormous speedups. However, large caches have certain important drawbacks, stemming from Titanium's barrier synchronization semantics. OrderCache buffers writes during computation, but must flush all dirty lines at barriers. Flushing a large OrderCache may be quite costly. First, each processor must scan all of its cache entry descriptors to find dirty lines. This may be time consuming if the cache so large as to cause virtual memory thrashing. Second, flushing a large number of dirty lines requires the issue of a large number of messages. This makes communications much burstier, particularly when many processors arrive at a barrier at the same time.

We have devised a microbenchmark to assess the cost of scanning a cache for dirty entries. Figure 5 illustrates the time required for processors to enter 1024 barriers in rapid succession, for OrderedCaches with varying numbers of entries. The time reported includes both the time to sweep the cache as well as the time to exchange messages to establish the barrier proper. This communication time is visible as the vertical offset between the times for varying numbers of processors. Sweep time as caches grow larger is visible as the upward trend in times toward the right of the graph. Performance is fairly constant for caches up to about 2^{10} entries. In the current implementation, each descriptor requires eight bytes. Thus, the descriptors for a 2^{11}-entry cache occupy sixteen kilobytes, which is the size of the L1 hardware data cache in each NOW UltraSPARC. Caches larger than this show a clear slowdown; by the time the cache contains 2^{14} entries, as much as three seconds is spent just sweeping through descriptors.

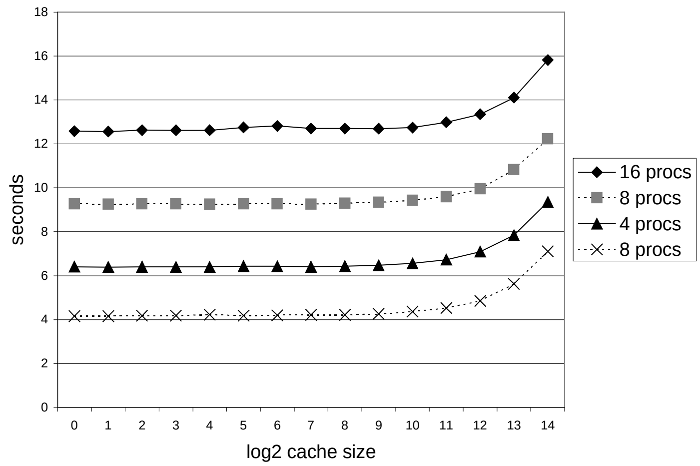

Figure 5: Cache size versus time to enter 1024 barriers.

3.3 Application Kernel: The Conjugate Gradient Method

The Conjugate Gradient Method (CG) is an iterative method commonly used to solve sparse linear systems. These systems actually appear often in many types of problems attacked by large parallel computers. Some examples are finite difference and finite element methods, which are used to solve problems in areas such as circuit and structural analysis [14].

The primary portion of the computation deals with a matrix-vector multiply. The multiply usually involves communication between processors to retrieve the matrix or vector components. A naïve method could simply execute a standard multiply. Unfortunately, this often leads to poor performance due to network latency and congestion. At this point, a lot of complexity can start to enter the algorithm as attempts are made to minimize communication and hand-optimize its overlap with computation. In the process, however, the original, easily understandable algorithm becomes clouded and hidden.

To test the effectiveness of the OrderCache, a naïve CG method on OrderCache was pitted against four different versions of CG running on the normal Split-C [2] Active Messages runtime (libsplit-c). A new version of the Split-C runtime (libOC-c) re-implemented certain calls to channel through the OrderCache. The mapping from Split-C language features into the libOC-c runtime is as follows:

Split-C notation	Split-C Terminology	OrderCache call
=	"read/write"	Blocking read/write
:-	"store"	Asynchronous read/write
:=	"gets/puts"	Asynchronous read/write

All 8-, 16-, 32-, and 64-bit versions of the libsplit-c calls were updated. Because OrderCache currently only handles values up to 32-bits, 64-bit calls were split into two 32-bit OrderCache calls. The synchronization functions were also updated to reflect this change. In order to ensure global synchronization at the relevant events, the entire libOC-c cache was flushed and invalidated at all synchronization events. Note that libOC-c preserves the OrderCache consistency model. This is a slightly stronger model than libsplit-c's, because makes stronger guarantees on write ordering.

Four versions of the CG method were analyzed, each slightly different in optimizing communication patterns:

- **Version 0 (v0).** This version is a naïve version. It does not attempt to optimize communication patterns and uses all blocking reads and writes. All pointers are global.
- **Version 1 (v1).** A "store" operation communicates values to other processors before each matrix multiply.
- **Version 2 (v2).** This version uses a blocking read/write, but attempts to do more things with local pointers than v0.
- **Version 3 (v3).** The "gets/puts" operations are used to move values to local memory during the computation.

All versions use a Split-C spread array to scatter the data over the network nodes. Only the version v0 was run on libOC-c. All versions v0, v1, v2, and v3 were run on the standard libsplit-c to compare against the libOC-c results. The tests looked for how well libOC-c performed against libsplit-c and the optimized versions given a specific problem size and number of processors. They considered how well each version scaled as the problem size and the number of processors were varied independently.

To determine how well libOC-c performed against the optimized versions, the cache size was tuned for individual runs. The 2-, 4-, and 8-processor tests used a cache size of (3, 6) (3 bits of processor number and 6 bits of address) or 512 cache lines. The 16-processor test used a cache size of (4, 6). This combination was found to produce good hit rates as well as low overhead in cache maintenance costs.

3.3.1 Scalability in Number of Processors

In this test, the problem size was held constant at a 128-square matrix, and the number of processors was varied from two to sixteen. The results are shown in Figure 6. The graph shows that libOC-c (and OrderCache) performs well compared to the different optimized versions as the number of processors was increased for this problem size. The scaling is on track with the v0, v2 and v3 versions of CG. Interestingly, the unoptimized v0 version performed well on both runtimes. This suggests that communications may not be the critical path in this test. This can also be seen in v3's only marginal improvement over v0. The v1 version, however, performs particularly poorly. This may be due to its large number of outstanding network requests. The overhead of these multiple requests could be slowing down this version.

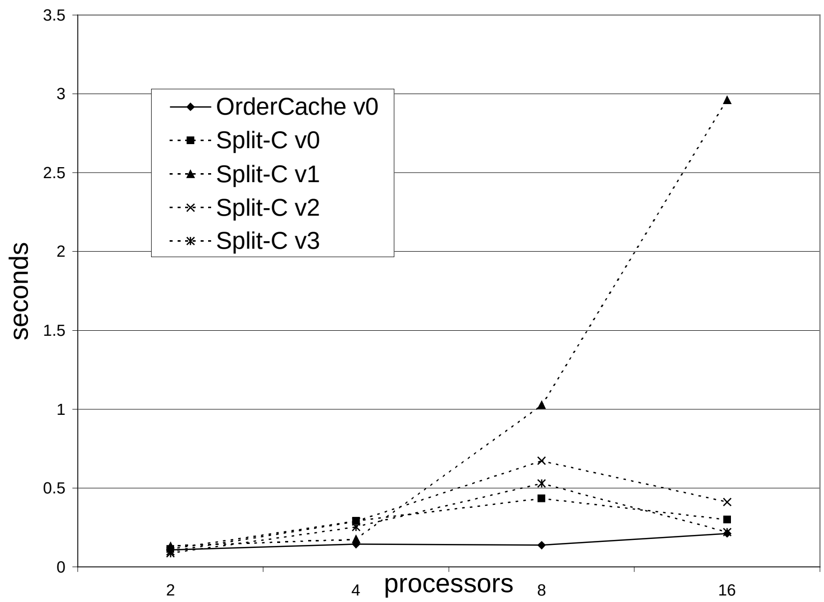

Figure 6: Conjugate gradient processor scalability

3.3.1.1 Cache Hit/Miss Rates in libOC-c

The read and write cache hit rates are shown below:

Number of Processors	Read Hits	Read Misses	Read Hit %	Write Hits	Write Misses	Write Hit %
2	0	0	0.00%	0	64	0.00%
4	3956	736	84.31%	0	32	0.00%
8	2438	1104	68.83%	0	16	0.00%
16	627	1064	37.08%	0	8	0.00%

As the number of processors increases, the rate of communications decreases, as is expected from the spread array for a fixed problem size. The cache hit percentage, however, declines significantly for the 16-processor test. This accounts for the relatively large increase in the time of the 16-processor test case.

3.3.2 Scalability in Problem Size

In a second test, the number of processors was kept constant at sixteen, and the problem size was increased. Figure 7 shows the results. Although the v1 version did poorly in the processor scaling test (and still does poorly for small problem sizes), it scales very well as the problem size increases. On the other hand, the other versions (v0, v2, and v3) perform very poorly on large problem sets. This may be due to the blocking reads and writes performed by v0 and v2. The v3 version does poorly because its asynchronous reads and writes overlap with only a limited amount of computation. The v1 version, however, seems to do a much better job of overlapping network requests with computation time.

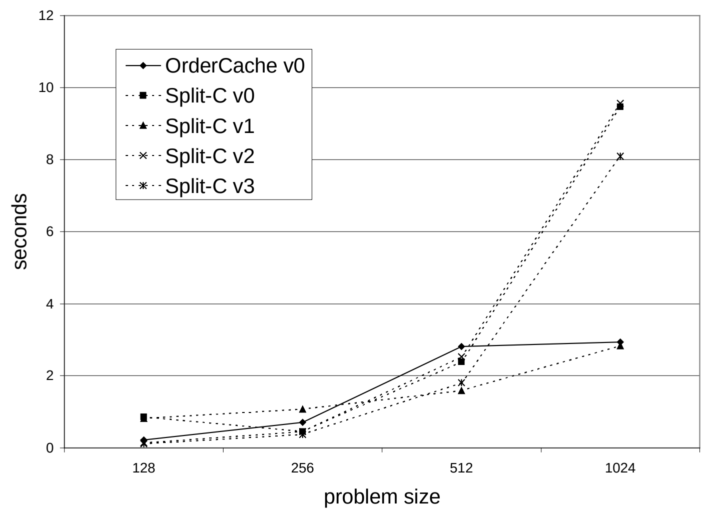

Figure 7: Conjugate gradient problem size scalability.

3.3.2.1 Cache Hit/Miss Rates in libOC-c

Why does libOC-c do well even though it uses blocking versions of reads and writes? Looking at the cache hit rates below, the percentage of hits increases dramatically with the problem size:

Problem Size	Read Hits	Read Misses	Read Hit %	Write Hits	Write Misses	Write Hit %
128	646	1064	37.78%	0	8	0.00%
256	5658	2576	68.72%	0	16	0.00%
512	32670	6048	84.38%	0	32	0.00%
1024	153323	12992	92.19%	0	64	0.00%

The cache is able to swallow most read network requests due to good locality in the code. Far fewer requests are sent, which greatly improves performance.

3.3.3 Ghost Nodes

A common practice when hand optimizing these algorithms is to create "ghost nodes" [2]. A processor explicitly communicates with other processors early on in the computation to prefetch the values it needs for this round. These are then stored locally and the computation proceeds. This is the root cause of much of the added complexity in these codes. In essence, libOC-c (and OrderCache) is an automatic mechanism for creating these "ghost nodes". By allowing the runtime to perform this task for the programmer, it may be possible to use simpler, more naïve algorithms, and still achieve performance comparable to more aggressively optimized systems.

4 Future work

The OrderCache is still a very immature system. There are still a large number of optimizations that can be done at both the compile-time and run-time levels. Because of the setup of the cache, OrderCache has the ability to recognize locality and generate bulk messages instead of individual requests. Additionally, adopting a proactive posture in cache flushing could help to lessen the network burstiness currently experienced at synchronization events. Help from the compiler in the form of profiling and cache page coloring [2] could also boost the effectiveness of the cache. In addition, both would aid in the determination of optimal cache sizes under various conditions.

When dealing with Clumps [14], networks of SMP's, the network problems become even worse as more processors are tied to fewer network interfaces. In this case, more aggressive caching and cache tuning would be necessary to alleviate these problems. It may be necessary to introduce additional multi-threaded support into the caching mechanisms.

Greater application control over caching policies may also be beneficial. Controlling cache replacement policies may help some applications that work better with set-associative or hashing caches. If an even weaker consistency model is sufficient, better control of the cache may help to avoid stalls. An innate advantage may exist relative to pure SMP hardware caches. Because the caching system is under software control, its use and its policies could ultimately be decided by the application itself, leading to better performance. This argument is similar in flavor to that of extensible operating systems, such as the Exokernel [12].

The Sequencer, although too strongly consistent for Titanium's needs, may yet prove useful in other language environments. A reordering rate of roughly one in two million compellingly justifies the deployment of a fast path for ordered messages. However, it makes it difficult to assess the performance impact when reordering does occur. The low reordering rate is likely the result of the use of Myrinet switches in the Berkeley NOW. Future deployments of Sequencer on other network hardware may allow us to explore this issue more deeply. Active Messages on ATM networks, for example, are known to have higher reordering rates [6]. One might also use randomized fault injection to artificially elevate the reordering rate on the current NOW network.

5 Related Work

There has been a lot of work across disciplines on cache coherence and consistency related issues. These range from multi-processor machines to file systems and the Internet. The C Region Library (CRL) showed that a high-performance software-only distributed shared memory system is viable [10]. The Stanford FLASH (with the SPLASH-2 benchmarks) and the MIT Alewife projects are attempting to address the cache coherence performance problems. Also, the Cray T3D uses a hardware version of a weakly consistent cache [1].

In the area of language design, Split-C [2] is a SPMD style programming language used in explicit communication programs. For Internet caches, a concept called BASE [10] was proposed to describe the weak consistency of these caches and how to deal with them. The Xerox-PARC Bayou project [2] proposed a weakly consistent database for mobile users, and the Coda project [12] allows weak consistency in file systems.

6 Conclusions

In a NOW, hardware support for consistency across processors is non-existent. Under these conditions, building strong consistency becomes very expensive, especially when communication costs are much higher than local memory accesses. A weak consistency model, on the other hand, can allow more efficient runtime systems to be built. One such system, a caching mechanism, is simple to implement and provides this weak consistency. In return, it allows the run-time system to handle naïve communication patterns with performance comparable to hand optimized patterns.

Experiments with the Conjugate Gradient Method suggest that instead of dealing with these complex program optimizations, the programmer can rely on the run-time system to help in the optimization of some tasks. This alleviates much of the coding burden from the programmer and leads to programs that are more understandable and easier to debug. If the performance boost from the caching effect is acceptable, then the tuned version may never need to be written. This effect has the potential to translate into shorter development times for these types of applications.

7 Acknowledgments

This paper was done with the support of the entire Titanium project team. Special thanks to Kathy Yelick and Arvind Krishnamurthy for their advice and pointers.

8 References

[1] Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy, Steve G. Steinberg, and Katherine Yelick, Empirical Evaluation of the CRAY-T3D: A Compiler Perspective, International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June 1995.

[2] E. Bugnion, J. Anderson, T. Mowry, M. Rosenblum, M. Lam, Compiler-Directed Page Coloring for Multiprocessors, Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), October, 1996.

[3] David Chaiken, Anant Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost, Proceedings of the 21st Annual Symposium on Computer Architecture, pages 314-324, April 1994.

[4] David E. Culler, et al., Parallel Programming in Split-C, Supercomputing, Portland, Oregon, November 1993.

[5] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, B. Welch, The Bayou Architecture: Support for Data Sharing among Mobile Users, © 1994 Institute of Electrical and Electronics Engineers. Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, California, December 1994, pages 2--7 (IEEE Computer Society Press).

[6] T. von Eicken, V. Avula, A. Basu, and V. Buch, Low-Latency Communication over ATM Networks using Active Messages, Presented at Hot Interconnects II, Aug 1994, Palo Alto, CA. An abridged version of this paper appears in IEEE Micro Magazine, Feb. 1995.

[7] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier, Cluster-Based Scalable Network Services, SOSP-16, 1997.

[8] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, Thomas E. Anderson., GLUnix: A Global Layer Unix for a Network of Workstations, to appear in Software Practice and Experience, 1997.

[9] J Gosling, B. Joy, G. Steele, The Java Language Specification, Addison-Wesley, Copyright © 1996 Sun Microsystems, Inc, 1996.

[10] Information Sciences Institute, RFC-793: Transmission Control Protocol, Defense Advanced Research Projects Agency, 1981.

[11] K. Johnson, et al., CRL: High Performance All-Software Distributed Shared Memory, SOSP 1995.

[12] Kaashoek et al., Application Performance and Flexibility on Exokernel Systems, SOSP, 1997.

[13] J.J. Kistler, M. Satyanarayanan, Disconnected Operation in the Coda File System, ACM Transactions on Computer Systems Feb. 1992, Vol. 10, No. 1, pp. 3-25.

[14] S. Lumetta, A. Mainwaring, D. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. Proceedings of SC97, San Jose, California, November 1997.

[15] A. Mainwaring. Active Message Application Programming Interface and Communication Subsystem Organization. University of California at Berkeley, Computer Science Department, Unpublished Manuscript, December 1995.

[16] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve and Tracy Harton, An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors, ASPLOS VII, 1996.

[17] J. Shewchuk, An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, CMU-CS-94-125, Carnegie Mellon University, 1994.