Distributed Shared Memory on IP Networks

CS 757 Project Report
Instructor: Mark Hill
University of Wisconsin-Madison Department of Computer Science

Students: Dan Gibson, Chuck Tsen
University of Wisconsin-Madison Department of Electrical and Computer Engineering
degibson@wisc.edu; chuck.tsen@gmail.com

Parallel programming is becoming an increasingly popular method of improving computer performance. Unfortunately, parallelized hardware remains expensive in the form of complicated symmetric multiprocessors and high-end server platforms. To provide a more available platform for parallel execution, we revisit the topic of implementing distributed shared memory on networks of commodity workstations. We discuss the implementation and evaluation of a distributed shared memory system, using existing Ethernet/IP-based networks for communication between logical threads. Our implementation leverages user-space programming primitives to provide a software-only solution for executing parallel programs on commodity hardware and operating systems. We demonstrate application-dependent speedup over uniprocessor executions for simulated workloads, even on small networks. We also observe the effect of heterogeneity in our network, and its significant performance impact.

1. Introduction

With commodity hardware becoming cheaper, and open-source, free operating system solutions such as Linux or FreeBSD gaining popularity, cluster computing has become commonplace for applications that exhibit large amounts of control parallelism. Concurrent execution of batch jobs, as in Condor [13], and parallel servicing of web and other requests [3] dominates the cluster industry, employing inexpensive systems to achieve very high throughput rates. Some cluster systems also employ otherwise idle cycles on workstations connected to a network to improve performance of batch, concurrent jobs.

While some workloads can benefit from concurrently running processes on separate machines with little or no communication, many workloads do not have sufficient explicit parallelism to exploit concurrency in this manner. Some of these workloads can achieve speedup on networks of workstations using other cluster technologies, such as the MPI programming interface [15]. Under MPI, machines may explicitly pass messages, but do not share variables or memory regions directly. For applications that do not easily lend themselves to the message passing programming model, the only other option for parallel execution is to run the workload on a large simultaneous multiprocessor computer, which supports shared memory.

As an alternative to this approach, we present an implementation of distributed shared memory (DSM) that operates over networks of workstations. Under our implementation, each node on a network can host a single thread that operates inside a larger pool of threads within a shared memory program. Memory is identically named at the level of C++ source code, with all sharing details hidden by the implementation. Additionally, we provide a mechanism by which the user can tune the sharing behavior and performance of the underlying DSM implementation by overtly specifying the sharing granularity. This gives the opportunity to tailor sharing parameters to a given workload, or working set size.

The challenges inherent in the creation of a distributed shared memory system over an IP network are great. Firstly, the semantics of IP networks allow messages (packets) to be dropped if the network becomes congested, which implies that any implementation must account for the loss of an arbitrary packet. Even the presence of QoS measures for network reliability cannot provide guaranteed packet delivery [7,18]. Only use of the TCP/IP transport protocol adequately hides packet loss, but the overhead of TCP/IP overhead is large, both in memory and execution time. Furthermore, IP networks can deliver packets any time after they are sent, subject to network availability and buffering at routers and switches. This policy differs greatly from an in-order, fixed-latency interconnect.

In addition to unreliability, IP networks are extremely latent compared to typical memory systems. Round-trip-times in the range of 100 μs are common in modern "fast" 100M Ethernet. Communication latencies can easily climb to hundreds of millions of execution cycles over IP networks.

The sustained bandwidth of IP networks also does not compare to typical memories; typically 100M Ethernet will support approximately 12.5 MB per second in a sustained point-to-point connection. This figure is separated by orders of magnitude from even mediocre commodity memories.

Perhaps most daunting, there exists no convenient, low-overhead mechanism to provide access to a consistent memory state on a remote machine, yet also preserve existing shared-memory programming binaries and source code functionality. The most often employed mechanism to accomplish this sharing is the use of page-level protection bits to automatically interpose on certain loads and stores to that page of memory. However, this dictates page-granularity sharing, ignores the effect of unshared variables, and has significant execution overhead.

Our solution accounts for these difficulties and demonstrates speedup among some simulated workloads, under both strict and relaxed consistency models. The remainder of this paper is organized as follows: Section 2 surveys prior work related to DSM over IP (DSM/IP); Section 3 describes the authors' implementation of DSM/IP; Section 4 outlines the experimental methods used in the correctness and performance evaluation of DSM/IP; Section 5 presents performance results; Sections 6 and 7 summarize our findings present concluding remarks.

2. Related Work

There have been many implementations of DSM on IP networks. A common approach is to use the virtual memory system to provide a level of consistency checking in hardware, employing system calls to protect page-sized regions of programs. Chief among the DSM/IP systems using this technique are Quarks (and its predecessor, Munin) [25], and TreadMarks [2]. The primary difference in these two implementations lies in their design philosophies—TreadMarks assumes a large computation overhead to attempt to reduce the IP communication overheads required by DSM. In line with this philosophy, TreadMarks uses the UDP transport protocol for communication. Quarks takes the opposite role, attempting to minimize the computational requirements of DSM/IP as much as possible, while allowing a greater degree of communication overhead. In contrast to TreadMarks, Quarks uses a connection-oriented communication scheme, TCP/IP. One of the greatest weaknesses of both Quarks and TreadMarks is the pre-defined granularity of sharing dictated by use of the virtual memory system (page-level sharing). IVY [12] is another example of DSM/IP employing page-level sharing, but IVY enforces a single-writer, sequentially consistent programming model, where other implementations allow multiple writers. Also falling into this category is StarDust [4], which, in addition to DSM, implements message passing [15].

Another popular approach to DSM/IP is in realm of operating systems. Plurix [20,27] from the University of Ulm is a Java-based operating system that supports DSM/IP natively for Java applications. The Java language is very conducive to DSM/IP, as it runs on a virtual machine monitor (VMM), which may change depending on the given operating system. Thus, it is possible for multi-threaded Java code to run largely without modification under Plurix. The Mirage system [8] is similar. It is integrated into an existing operating system, and employs page-level sharing.

The Brazos project at Rice University [22,23] represents another class of DSM over IP implementations—those that exist entirely in user program space. Brazos executes on Microsoft ® Windows ® -based machines (as does an implementation of TreadMarks [2]) in a multi-threaded environment designed to hide communication latencies. In place of IP broadcast, Brazos uses IP multicast, which has a slightly higher performance overhead than broadcast packets, but with the added benefit of improved scalability. This also mandates the use of TCP/IP connections in Brazos.

The SHRIMP multicomputer [6] is another example of commodity hardware used for DSM, but SHRIMP does not use an IP network. Instead, it uses a SAN-like custom network to produce a more closely coupled cluster of independent x86/Linux based systems. Communication among these nodes is based on the stream programming model, but has significantly less overhead than streams over IP networks.

The consistency model varies widely among the systems listed above. Popular among these is lazy release consistency, used by TreadMarks [2] and others. Under this scheme, multiple writers to a given memory location are allowed, provided a final memory image is eventually built from the contributions of all writers. Several systems have successfully implemented sequentially consistent DSM, including IVY [12] and Plurix [20,27]. Quarks [25] uses a configurable consistency model, allowing release or sequential consistency.

Our technique of syntax modification is similar to the Check-In/Check-Out annotations as described in [10]. CICO semantics require the programmer to specify when memory regions are expected to be accessed, and what type of accesses will occur (read or exclusive). These annotations enable simpler coherence protocols, but never change the correctness of a program. They merely serve as "hints" that accesses of a certain type may occur in the near future, or that accesses to a particular region will not occur for some time.

Our implementation also shares some similar features to the Tempest project [10,19]. Under Tempest, the programmer makes use of explicit coherence organization, and is also provided message-passing primitives. Tempest's fine-grained access control and user-defined permissions allow a great amount of customization of sharing patterns among processors. Alternatively, Tempest allows programmers to use already written coherence policies, to reduce programming complexity. Tempest is a description of an interface, that does not assume any particular implementation. One such implementation is Typhoon, described in [19].

In general, all of these DSM/IP systems have shown speedup for some classes of applications—those that tolerate communication latency well, and have course-grained sharing between threads.

3. DSM/IP Implementation

We present a software-only realization of distributed shared memory, implemented as a user-level library. In our system, each node maintains a copy of each shared memory region at all times. At times, some portions of shared memory may be inaccessible, due to coherence and consistency requirements. In general, shared regions are not page-aligned, and can be of arbitrary size. The granularity of sharing can be selected by the programmer if desired as an optimization to their DSM-enabled software, or assigned automatically to optimize network traffic.

3.1 API Overview

Our implementation changes the manner in which shared memory regions are declared and accessed, using a very simple set of accessor and mutator methods. Each shared memory region is declared as a C++ object, derived from a single `SharedObject` class. Children of `SharedObject` include `SharedInt`, `SharedIntArray`, `SharedFloatArray`, etc. Each object can be accessed through accessor function `Read()` or mutator function `Write()`. Thus, the *syntax* by which shared memory is accessed is different than that of normal variables in C++, but the *semantics* of read and write accesses are not changed. Each `Read()` and `Write()` operation is atomic, and behaves according to the expectation of the programmer. Note that these operations do not provide any additional synchronization—though primitives are provided for synchronization (below). The use of explicit access and mutate functions allows user-level code (within our library) to perform coherence checks and any required network communication without intervention from the OS. We note that it is possible to hide some uses of the accessor and mutator functions through use of operator overloading in C++, which would unify some syntactic differences. We have not implemented this feature, as it would not be possible to provide both identical array reading and array writing syntax using this method, due to conflicts in the use of the `[]` operator in C++.

At declaration points of shared memory regions, most object types allow the user to specify the granularity at which coherence information should be maintained. This granularity determines the *segment size*, the size of the region protected by a single coherence state variable. The segment size may vary from object to object, at the whim of the programmer. In order to ensure that coherence messages are not too large, the DSM implementation may increase the granularity of sharing, to improve performance. We have not explored the effect of variable-sized coherence granularity in our system's performance, but we expect that there will exist an optimal sharing granularity on a per-workload, per-data set basis.

In addition to providing shared access to memory regions, our API also provides lock and barrier synchronization primitives. The `DSM_Barrier()` function guarantees that no thread executes subsequent instructions until all threads have arrived at the barrier. Class `DSM_Lock()` provides mutual exclusion functionality, with the usual lock acquisition and release semantics.

As with many APIs, function calls are required to initialize the multithreaded environment and to gracefully exit the program. After the initial setup call, `DSM_Startup()`, the requested number of threads begin execution immediately at the return point of the function, synchronized to within a single barrier delay. At that point, the only perceivable difference between logical threads (aside from residing on separate machines) is the value returned by `DSM_Startup()`. This value is a unique thread identifier, between zero and the number of threads minus one, inclusive.

3.2 Library Implementation

The DSM library is two-layered; the lower layer is an abstraction built from available UDP networking primitives, and is used by the upper layer for all inter-machine communication—we refer to this layer as the Communication Backbone (CB). The upper layer is the Coherence Engine (CE), which implements the shared-memory functionality using the primitives provided by the CB. The `Read()` and `Write()` functionality is provided by the coherence engine, while all other API

(synchronization and startup) is built directly into the communication backbone. It is the coherence engine that determines the consistency model provided by the DSM implementation.

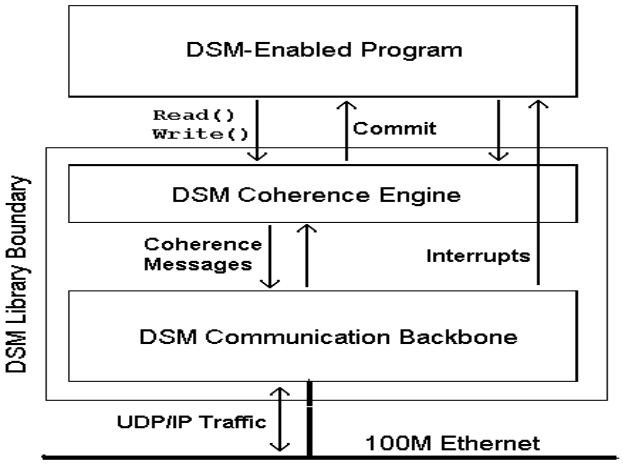

Figure 1 - Interaction of engine, backbone, and user program

3.2.1 Communication Backbone

The CB is the means by which all inter-machine (inter-thread) communication occurs. It is built from the UDP/IP networking interface, and is optimized for round-trip time (RTT) on a local area network (LAN). The CB extends the functionality of UDP to include guaranteed message delivery and at-most-once message delivery. Thus, the CB abstracts the IP network as a much more reliable interconnect. This added reliability is necessary to provide arbitrary program consistency models.

Upon receipt of a message, the CB delivers the received message and all data associated with it to a specified handler function, which is similar to an asynchronous interrupt. These handlers include built-in functions in the CB to handle certain classes of message, and vectored handlers for each of the message types used by a given CE.

The CB enables messages to be passed between threads using only the logical thread number as a destination address. That is, the CB maintains a list of thread identifier to IP address/port mappings, which is established in the initial system synchronization phase. The details of network communication are hidden from the implementation of the coherence engines to provide modularity and to enable changes in the CE without reworking the CB. Below, we outline how the backbone provides these abstractions.

3.2.1.1 Packet Formatting, Thread-ID Addressing, Vectored Interrupts

The CB operates on top of the usual UDP/IP interface, using the `recvfrom()` and `sendto()` system calls. The CB forms an additional layer of abstraction, including its own specific message header fields in each packet. Those fields are:

- `progID`: A unique 16-bit integer associated with each DSM-enabled program. This number is agreed upon out-of-band, in a globally visible `dsm_setup` file. Use of this identifier allows, in theory, for multiple DSM-enabled programs to run concurrently, and prevents latent messages form previously run programs from affecting current executions.
- `msgType`: A 16-bit integer specifying the message type associated with a given packet. This value is used primarily to vector interrupts to the CE and other supporting software. The upper three bits are reserved for designating ACK, ACK_REQUESTED, and NACK (note: no CE implementations currently use NACKs).
- `dataLen`: A 16-bit integer specifying the length of accompanying data in bytes. `DataLen` is used for bookkeeping and the mechanics of the UDP/IP library interface.
- `retransNum`: An 8-bit integer specifying the number of retransmissions allowable on this message. No implementations currently employ this field; the default value is five.
- `frag`: An 8-bit integer indicating the fragmentation state of packet, used for reassembly. Note that sub-fragmentation is never needed, but fragmentation may occur if sending more than the number of bytes allowable by UDP/IP (1450).
- `seqNum`: A 64-bit sequence number field. Each thread has 2^{64} / N sequence numbers reserved for that thread, where N is the number of threads in the system. Therefore, for our target systems (*N* on the range [1,128]), each thread can send around 144 *quadrillion* packets. Given a typical round-trip-time (RTT) of 0.1ms, this represents over nine million years of continuous packet transmission. Hence, we do not check for the case of wrap-around of sequence numbers.

Each of the above fields is filled in automatically by the CB, with the exception of the `dataLen` field, which is filled in by the CE. The final formatting is shown in Figure 2.

Ethernet	IP	UDP	DSM	Payload	CRC
22 B	26 B	8 B	16 B	Up to 1450 B	4 B

Figure 2 - DSM/IP Packet Format

The CB also provides a mapping of thread-identifier-to-IP-addresses, that allows messages to be passed among threads, identifiable by logical thread number, instead of by specifying IP address and port information. This mapping becomes available on the receipt of the first message from a given thread, namely the DSM_JOIN message. This typically occurs during the HELLO phase of initial synchronization, described in subsequent sections.

On each `SendMessage()` call or receive message event, the address mappings are accessed to determine either destination address or sender thread identifier. That information is then passed to the vectored event handler for message receipt, or the UDP/IP interface via `sendto()`. This information is reflected in the corresponding packet's IP and UDP message headers, respectively. DSM relies upon the Linux operating system to cache physical Ethernet (MAC) addresses for machines, or to use the address resolution protocol (ARP) as needed [18].

When the CB receives a packet from the network, the backbone determines the message type by use of `msgType` field in the DSM header. Based on this type, the CB then calls a specific message handler to handle the message. Often, this message is delivered to the CE, and results in some change in the coherence state of the local machine. In other cases, the message is intended for the synchronization API functions, or could be part of the initialization scheme. In general, message handlers cannot be interrupted by subsequent messages—instead, interrupts are deferred until after the handler completes. However, network polling still occurs within `SendMessage()` primitives, even in event handlers, to prevent deadlock.

3.2.1.2 Startup and Synchronization

Before the parallel phase of a DSM-enabled program may begin, each thread must initiate communication with other threads to determine IP/port mappings, ensure the correct number of machines are participating, and establish synchronization between threads. All of this functionality is implemented in the CB, via the API call `DSM_Startup()`. This call takes as parameters the argument stream of the currently running process and the desired number of processors to use for the workload. After `DSM_Startup()` returns, the requested number of processors are logically available, and each processor has logically "just exited" from `DSM_Startup()`.

On entrance to this call, the local thread broadcasts a DSM_JOIN message on the network. This is a unique broadcast message that expects only one response, from the first DSM process to enter the network. This process is always mapped to thread identifier zero, and serves a special "server" purpose during initialization and during error conditions. Thread zero is otherwise in a peer-to-peer relationship with all other threads during normal execution.

If the "server" thread responds, the response includes the local machine's thread identifier for this execution and thread zero's IP address/port information. The local machine can then enter the HELLO phase, which is described in detail below.

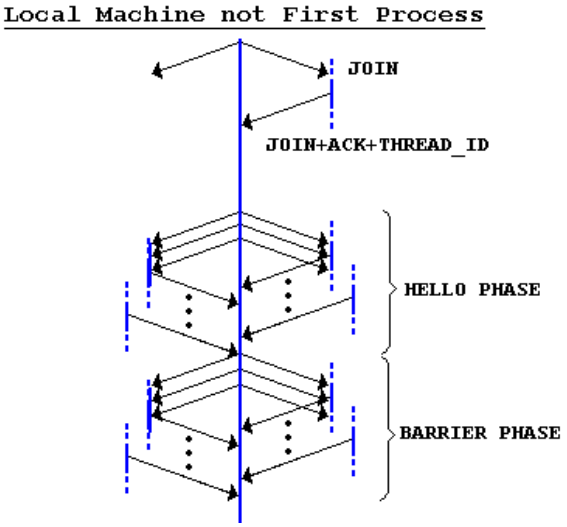

Figure 3 - Communication pattern of all non-zero threads during initialization

If the server thread does not respond after a fixed number of randomized exponentially longer timeouts (totaling approximately 1.25 seconds), the local process assumes it is the first DSM-enabled process on the network. It then enters server mode, and assumes thread identifier zero.

Immediately after entering server mode, thread zero broadcasts a single DSM_WAKEUP message on well-known port. This message contains the number of threads requested, the name and program identifier of the binary to be executed, and the argument stream to the binary. We implemented a useful process-spawning daemon to listen for DSM_WAKEUP messages, and automatically `fork()` and `exec()` DSM-enabled processes as needed by a new "thread zero." These new processes arrive at their subsequent `DSM_Startup()` calls, and eventually

send DSM_JOIN messages, to which thread zero then responds. Figure 4 corresponds to the execution of the first thread's initialization communication pattern in a DSM-enabled network.

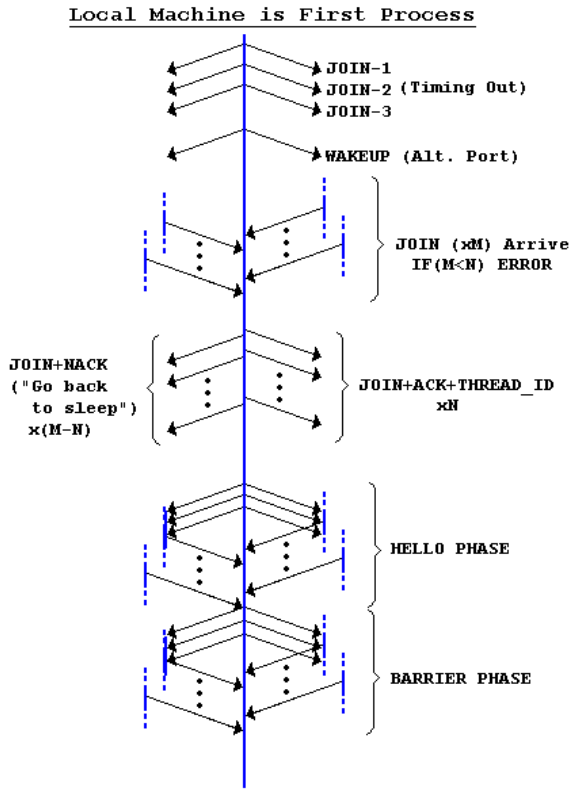

Figure 4 - Communication pattern of thread zero during initialization

Note that more than the necessary number of threads may respond to DSM_WAKEUP with DSM_JOIN. In general, for a request of N threads, there will be M DSM_JOIN messages inside of a small timeout period. If M<N, then an insufficient number of threads have joined the system, and the API startup call fails. If M≥N, thread zero sends thread identifiers inside DSM_JOIN_ACK messages to the first N threads to respond. Subsequent DSM_JOIN messages (M-N in total) will receive DSM_JOIN_NACK responses, in which case the receiving process exits under normal conditions, as it is unneeded to fulfill the requested number of threads.

Once at least N threads have been issued thread identifiers, thread zero initiates the HELLO phase (below). From the above initialization, all threads have determined mappings for their local machines and thread zero's machine, but do not have mappings for other threads. The purpose of the HELLO phase is to "introduce" each thread to all other threads, through a series of handshakes.

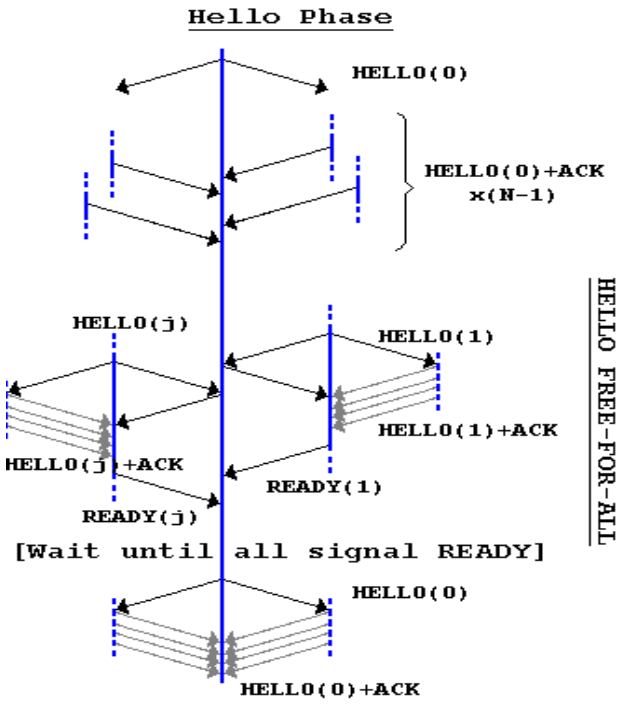

Figure 5 - HELLO phase operation

Thread zero initiates the HELLO "free-for-all" with a special HELLO(0) message, which signals to all threads that all other threads are ready to begin the HELLO phase. After receipt of the first HELLO(0) message, all threads then broadcast their own HELLO messages, roughly simultaneously. Each of these messages reaches all other threads, thereby creating an IP/port mapping on receipt. Once all N mappings are present on the local machine, a READY(j) message is sent to thread zero, and the local thread spin-waits until receipt of a second HELLO(0) message. This second message is generated to signal the end of the HELLO phase, and is sent when all threads have valid IP/port mappings for all other threads in the system. At this point, it is possible to send arbitrary messages between threads. After completion of HELLO, threads enter a `DSM_Barrier()` synchronization function, to ensure that all threads leave `DSM_Startup()` at approximately the same time.

3.2.1.3 Synchronization API Implementation

Two synchronization primitives are implemented within the CB: `DSM_Barrier()` functions and `DSM_Lock` objects. Barrier functionality is realized using one global integer at each node for each logical thread, indicating the last barrier at which a particular node has arrived. Barrier arrivals are broadcast to all nodes on the network—as soon as all threads are observed at a given node to have arrived at the next logical barrier

number, the local thread exits the barrier. Under this scheme, an invariant exists such that all logical barrier numbers in the system differ at most by one at any time.

`DSM_Lock` objects have two methods: `Acquire()` and `Release()`. A delicate correctness issue arises in the implementation of these functions, and is related to similar issues arising from the implementation of sequentially consistent coherence engines. The `Acquire()` and functions `Release()` must be *interruptible*, as at any time the CB may handle an incoming message, which could include attempts to acquire or release a lock. Specifically, there must be no opportunity for an incoming DSM_LOCK_ACQUIRE message to enable two or more threads to acquire a lock simultaneously. Our general technique to avoid this class of problem is presented in subsequent sections, with emphasis on sequential consistency concerns relating to coherence engine realizations and race conditions.

3.2.1.4 Guaranteed Message Delivery

The communication backbone provides guaranteed delivery in the same manner as the TCP/IP transport protocol—through the use of explicit acknowledge messages. The presence of these messages is *mostly* invisible to the coherence engine, so one-way communication appears as a single message. In general, ACK messages are handled entirely in the CB, without calling any handlers in the CE. However, since many coherence messages already follow the request/response model (i.e. a message requesting data), the CE may opt to use the ACK message to carry data. This data is not in turn protected by a subsequent ACK message—however, failed delivery of an ACK within a timeout period generates a retransmission. [18] points out that retransmissions are a potential source of multiple message deliveries. This concern and potential side effects are addressed in subsequent sections. As the case of message loss is very small under UDP/IP, this code addressing this case need not be extremely efficient, it need only ensure that in the event of packet loss, correct semantics are maintained. If guaranteed delivery is enabled for a particular message, the `SendMessage()` call to the CB blocks until receipt of all expected ACK messages.

UDP/IP was chosen primarily for its lower overhead as compared to TCP/IP [7,18,21], though UDP/IP also allows us to use ACK messages to carry data, which greatly decreases network response times. This advantage is critical to network performance, as under TCP/IP at least four packets would be required to send round-trip messages, discounting connection startup overhead. As a final added bonus, UDP/IP is a stateless protocol, which presents a simple "datagram" service. TCP/IP is connection-based, which would require each node to maintain connections to all other nodes, which would limit system scalability [9,18].

3.2.1.5 At-most-once delivery

In addition to guaranteed delivery, the CB also provides at-most-once delivery. The intersection of these services is guaranteed-once delivery, the semantics often used in traditional SMP systems.

To provide this and other services, the communication backbone implements additional header fields in addition to the usual UDP/IP header information. At-most-once semantics are easily achieved provided:

- No two packets with equal `seqNum` fields are ever processed (that is, messages already processed should be ACKed but otherwise ignored).
- All packets (including ACK packets) have a unique sequence number.

The CB ensures both of these points, by tracking the last sequence number received from each host. Incoming messages with lower sequence numbers have already been processed. While it is possible for disagreement to exist between hosts about what a particular host's next sequence number is (due to unicast messages), it is permissible to allow each host to track sequence numbers without synchronization.

3.2.2 Coherence Engines

We have implemented three different coherence engines (CE's) using the communication and synchronization primitives provided by the communication backbone. Each CE is described below in detail. We have included both sequentially consistent engines and an engine that leverages weaker consistency models for improved performance. Performance of the three engines is discussed further in section 5.

3.2.2.1 ENGINE_OU: Naïve Sequential Consistency

Sequential consistency as defined by Lamport [9] mandates that a total ordering of all reads and writes to a specific memory location must exist and correspond with the memory state of all participating threads, and this interleaving must maintain program order. This can be implemented by servicing all read and write events to a particular location in the order they are received, one-at-a-time, provided they are presented in program order. As our DSM implementation is software-based, all read/write events already occur in-order—ENGINE_OU simply ensures that all reads and writes are serviced one at a time and at the same location.

Specifically, ENGINE_OU assigns to each shared segment a permanent thread owner, evenly distributed across all threads. Each access (`Read()` or `Write()` call) to that particular segment must communicate with the owning thread, unless the owning thread resides on the local machine. In this manner, a total ordering of loads and stores is generated at the segment's owner. ENGINE_OU is very communication intensive, as nearly every access will generate network traffic.

3.2.2.2 ENGINE_NC: Ignoring Consistency

ENGINE_OU represents an extreme, literal implementation of sequential consistency. ENGINE_NC follows the reverse philosophy: ignore all consistency models and simply make accesses as fast as possible. Under this engine, no coherence state is maintained whatsoever. All `Read()` calls are serviced locally, using whatever value is present in local memory. All `Write()` calls are simply broadcast to the network as notifications of updates—ENGINE_NC disables the CB's guaranteed delivery mechanism to make the `SendMessage()` primitive non-blocking (and fast). As a result, not only is there no ordering of memory accesses, but certain processors may *never* become aware of all other processor's writes (due to lost packets), and final memory state is not guaranteed to be consistent across all machines. This extreme form of non-consistency goes beyond the release consistency model presented in [1], earning the engine the name *engine non-consistent,* hence ENGINE_NC.

It is important to note that no use of `DSM_Lock` and `DSM_Barrier()` primitives is sufficient to upgrade the consistency of ENGINE_NC. Since ENGINE_NC allows coherence messages (write updates) to be lost, no guarantee exists that memory will ever reach a consistent state across all machines.

3.2.2.2 ENGINE_MSI: Smarter Sequential Consistency

Seeking stronger semantics than those of ENGINE_NC, but better performance than ENGINE_OU, we implemented an engine based on the MSI coherence protocol, as described in [26].

The basic MSI protocol uses three states to track memory read or write permissions at each node: MODIFIED (`M`), SHARED (`S`), and INVALID (`I`). If a block of shared memory is in the `M` state at a given node, it has both read (`Read()`) and write (`Write()`) permissions to the data. Memory in the `S` state has read permissions associated with it, but writes are not allowed. Nodes with memory in the `I` state have neither read nor write permissions for their data. If a node needs to perform an operation for which it has no permissions, it must block and send a message over the network. When it has received the proper number of responses, it can upgrade its permissions and proceed with its operation. The following invariants always hold for the MSI CE:

1. If any node is in the `M` state for a given segment, all other nodes are in the `I` state for that segment.
2. There is always at least one node in either the `M` state or the `S` state.

We enhanced the MSI coherence protocol to make it run more smoothly on our DSM/IP system. The main goal we had in mind when making enhancements was to ensure consistency, given the semantics of our communication backbone. To achieve this, we keep track two extra pieces of "distributed-directory-style" information: 1) the number of total sharers of a given grain of data, and 2) whether a node exists somewhere on the network in the `M` state—we refer to this node as the "owner." This knowledge updated at each node when handling messages, and it is used to ensure the consistency of our implementation.

To demonstrate how ENGINE_MSI saves bandwidth and how the "distributed-directory-style" information augments our protocol, consider a node which has a shared variable in the `I` state and needs to obtain read permissions by upgrading to the `S` state. First, the upgrading node must broadcast an upgrade request to all nodes. Every other node will increment the number of sharers for the segment, but will respond with a message only if it has the segment in the `S` state. The upgrading node waits for n responses, where n is the number of sharers for the object as traced by the upgrading node. The upgrading node also checks that each sharing node returns a consistent value. We could eschew this check to improve performance; we did not remove the check as additional, dynamic verification of invariant 1, above. Once the requesting node has received shared permissions, the data may be read (via `Read()` calls) without sending any additional messages over the network, which significantly reduces network traffic.

To explain the necessity for keeping distributed-directory-style information about segment "ownership," consider the case when one node has data in the `M` state, and another node needs to obtain read permissions (upgrade from `I` to `S`). To accommodate this change, the bookkeeping for the number of sharers must increase from 0 to 2 at each node. To make this transition, each node must have knowledge of whether a segment is shared (in the `S` state remotely) or "owned" (in the `M` state

remotely). This extra bookkeeping is necessary due to the unordered interconnect presented by the CB.

3.2.3 Consistency Races

The communication backbone leverages the user-space interrupts provided by the Linux operating system to provide a mechanism by which incoming messages from the network can be serviced. The single thread operating on the local node does not observe this transparent interruption, but the DSM implementation must be aware of its effects. One concern is that coherence permissions may change shortly after a permissions check (as soon as a single instruction after), thereby raising concerns about consistency. Since the communication backbone does not guarantee that implementations of `Read()` and `Write()` are atomic, they must account for this phenomenon (Note that the CB *does* guarantee atomicity of interrupt handlers). Consider the code presented in Figure 6, which is a naïve realization of the `Write()` function and invalidate message handler for the MSI-based coherence engine (the OU engine uses a similar strategy). For illustration purposes, let us assume that the functions `AcquireM()` and `SendMessage()` are atomic (In practice, `SendMessage()` is fully interruptible but is not interrupted in this case because it is called from an interrupt handler. Regardless, this interrupt can be modeled as interrupts placed immediately before or after calls to `SendMessage()`. Furthermore, the implementation of `AcquireM()`'s functionality can simply use a heavyweight mechanism to ensure consistency, such as the disabling of interrupts.). This example applies to other interrupt handlers that change coherence state, as well as to the invalidation handler as outlined.

```
  void Write(int newValue) {
1   if( coherence != MSI_M )
2     AcquireM();
3   x = newValue;
4   return;
  }

  void Inv_Handler() {
5   coherence = MSI_I;
6   SendMessage(INVALIDATED);
7   return;
  }
```

Figure 6 - Naive implementation

Clearly, the intent of the `Write()` function is to ensure the variable `x` is in coherence state `MSI_M` at the time of the write. However, observe that an interrupt by a competing `Write()` call on a remote machine could generate an invalidation message interrupt at program point 3. The resulting ordering of statements is [1-2][5-7][3-4]. This interrupt leaves the `x = newValue` statement unexecuted, while the function `INV_Handler()` gives up write permissions for variable `x`. After the interrupt returns, the write to variable `x` occurs while `coherence == MSI_I`. This is a clear violation of the desired consistency semantics.

However, correct semantics are still attainable, allowing the `Write()` function to be interrupted. Figure 7 illustrates modifications to both `Write()` and `INV_Handler()` to ensure the desired invariants—that is, 1) writes occur exactly once, and 2) writes occur only in states that allow writing (`MSI_M` in this example).

```
  void Write(int newValue) {
1   g_temp = newValue;
2   g_flag = 1;
3   if( coherence != MSI_M )
4     AcquireM();
5   if( g_flag == 1 )
6     x = g_temp;
7   g_flag = 0;
8   return;
  }

  void INV_Handler() {
10  if( g_flag == 1
        && coherence == MSI_M) {
11    x = g_temp;
12    g_flag = 0;
    }
13  coherence = MSI_I;
14  SendMessage(INVALIDATED);
15  return;
  }
```

Figure 7 - Improved implementation

The improved implementation above ensures both invariants. Consider the following per-case illustrative proof (assuming an interrupt occurs sometime during the execution of `Write()` that invokes `INV_Handler()`):

Case 1: [1][10-15][2-8]

Assuming `g_flag == 0` *before* `Write()`, `newValue` *is unchanged and* `coherence != MSI_M` *at line 3, thereby write permissions are acquired at 4. Variable* `g_flag` *remains unchanged (1), so the write occurs at point 6 with invariants 1 and 2 maintained.*

Case 2: [1-2][10-15][3-8]

In the interrupt handler, `g_flag == 1` *implies that a write was in progress. The handler finishes the write at points 11 and 12. Coherence permissions change, and*

control returns to `Write()`. `Write()` *then acquires* `MSI_M` *permissions (needlessly, as the write has already logically occurred during the previous* `MSI_M` *permissions epoch).* `Write()` *observes* `g_flag == 0`*, therefore it should not again write to the variable. This case exhibits pathological performance, as extra permissions are acquired but not used.*

Case 3a: `coherence == MSI_M` *[1-3][10-15][4-8]*

`Write()` *observes* `coherence == MSI_M` *at 3. Immediately after, within* `INV_Handler()`*, the write is completed on 10-12 as in Case 2. Upon reaching 5,* `Write()` *observes* `g_flag == 0`*, thus the write has already occurred.*

Case 3b: `coherence != MSI_M` *[1-3][10-15][4-8]*

`Write()` *observes* `coherence != MSI_M` *at 3. Immediately after, within* `INV_Handler()`*,* `coherence != MSI_M` *implies no write occurs. Permissions change, the handler returns.* `Write()` *observes insufficient permissions to continue, acquires new permissions, and completes the write.*

Case 4: [1-4][10-15][5-8]

In this case `coherence == MSI_M` *after 4.* `Write()` *continues as in 3a.*

Case 5: [1-5][10-15][6-8]

`Write()` *has observed* `g_flag == 1`*.* `INV_Handler()` *observes* `g_flag == 1`*, completes the write and relinquishes permissions. Upon returning to* `Write()`*,* `x` *is again written. This cannot violate consistency, because writes to variables are not broadcasted, but lazily updated (under sequentially-consistent engines). Since no corresponding* `Read()` *call could have interposed, statement 6 does not affect the value of* `x`*. Hence, either the invariant* `g_temp == newValue` *or* `g_temp == x` *holds at statement 6.*

Case 6: [1-6][10-15][7-8]

`Write()` *has written to* `x`*.* `INV_Handler()` *observes* `g_flag == 1`*, and repeats the write. As in Case 5, this write is redundant and cannot violate consistency. Permissions change,* `INV_Handler()` *returns.* `Write()` *completes, and returns.*

Case 7: [1-7,8][10-15][8]

Logically, `Write()` *has completed.* `INV_Handler()` *relinquishes permissions and returns.* `Write()` *returns.*

For all cases, the write occurs logically once (or subsequent writes are guaranteed to have no effect), and each write occurs while coherence state is `MSI_M`. The technique outlined here is used in both the ENGINE_OU and ENGINE_MSI CE's (note that the implementation of ENGINE_NC need not consider this race), and in the `DSM_Lock` object realization.

4. Methodology

To evaluate our DSM/IP implementation, we executed our software on four heterogeneous x86/Linux-based machines. To verify the correctness of our implementation over the network, we developed a small suite of microbenchmarks. Next, to evaluate performance across applications with varying communication demands, we developed three simulated workloads. We tested each of the three engines outlined above in the context of these workloads. Finally, for each engine/workload pair, we varied the number of machines in the cluster to analyze the performance of different network configurations.

We originally intended our implementation to operate on medium-sized networks of up to 128 workstations. Unfortunately, we were unable to secure access to any such networks due administrative difficulties. Our DSM/IP implementation overwhelms a LAN to the point of suppressing TCP/IP streams due to the large number of UDP packets, which is undesirable for any public cluster. The network administrators we consulted refused to allow us access to their machines for this reason.

This setback allowed for the investigation of heterogeneity in DSM/IP. The heterogeneity of our network enabled us to evaluate the effects of adding a slow computer to a fast cluster. Two questions raised by heterogeneity are 1) Does adding a computer always contribute to improving the performance of a network? 2) If not, what is the threshold above which a computer must perform to for it to be beneficial? Furthermore, it is important to see how this answer varies across simulated workloads.

To gather information, we utilized event counters and cycle timers. By inserting counters into our code, we could record important events, such as the number of messages sent, the number of messages received, the number of timeouts, *et cetera*. Furthermore, we used high-granularity cycle timers to determine latencies of key events, such as barrier latency and the time spent waiting for responses to coherence requests.

4.1 Microbenchmarks

We used three microbenchmark to test for correctness of our implementations. Since we had several coherence

engines and unpredictable network behavior, the microbenchmarks were essential to ensuring correct functionality of our implementation.

The first microbenchmark, `message_test`, attempts to overload the network backbone and produce incorrect execution. From each thread, it sends a specified (large) number of messages to the next thread in rapid succession. This has the effect of generating many more messages than might be expected for the same number of threads under normal execution, and empirically demonstrates the backbone's ability to handle large numbers of messages. The second microbenchmark, `barrier_test`, iterates over a specified (large) number of barriers to stress the `DSM_Barrier()` mechanism. This puts maximal pressure on barrier synchronization, as the previous microbenchmark did for simple packet handling. The third microbenchmark, `lock_test`, puts maximal pressure on the `DSM_Lock()` primitive by using all threads to acquire the lock and increment a global variable simultaneously. If the configuration of the network passed these three tests for large numbers of iterations, we assumed the configuration was sufficiently functional to use these primitives to build more significant workloads.

4.2 Simulated workloads

We developed three simulated workloads to evaluate our DSM/IP implementation and to analyze it for performance. The goal in developing the simulated workloads was to represent workloads that might be run in a multiprocessor environment. Furthermore, we chose three applications that had varying levels of inter-node communication. It is worth mentioning that our implementations are relatively simplified, to facilitate quick development.

The first workload we developed, `sea`, had the highest level of inter-node communication. `Sea` is an integer version of the popular OCEAN benchmark from the SPLASH-2 suite [28]. A large array of integers is used to estimate the temperature at a given locale in the sea. As in OCEAN, we iteratively take averages of the adjacent points in the sea in each direction. For synchronization, we use the heavy-duty `DSM_Barrier()` between iterations. The frequency of interactions, and the division of sea locales across the network generate large amounts of network communication for this workload.

Our next workload, `genetic`, is a typical iterative genetic algorithm, which represents a moderate level of network communication. Using a distributed genetic process to "breed" a specific integer from a random population, it iterates in two phases. In the first phase, we determine potential solutions from the most fit members of the population, according to an evaluation metric. This metric contributes to the "think time" of `genetic`, as it does not access shared elements while evaluating a population member. We opted to keep this think time shorter than might be expected in a typical genetic process, to stress network latencies. In the second phase, we remove the least fit solutions from the population. We use `DSM_Barrier()` for synchronization, as there is still a significant amount of network traffic, and the two phases require changing reader/writer roles between threads.

Our final workload, `xstate`, is an exhaustive solver for complex mathematical expressions. It is a compute-intensive integer solver to find the zeroes of arbitrary functions. It has the lowest level of network communication, as most of the calculation is accomplished without need for communication across machines. We use a single `DSM_Lock()` primitive to protect a globally shared array of solutions to the multidimensional analysis performed by `xstate`, and a single barrier for end-of-execution synchronization.

4.3 Timer Validation

High-resolution counters were used extensively in this project, provided by the Intel ® x86 instruction `rdtsc`. This instruction provides a 64-bit cycle-accurate count of elapsed clock cycles since the counter was last reset. Timer validation was performed using more accessible but less accurate system calls, namely `gettimeofday()` and `sleep()`.

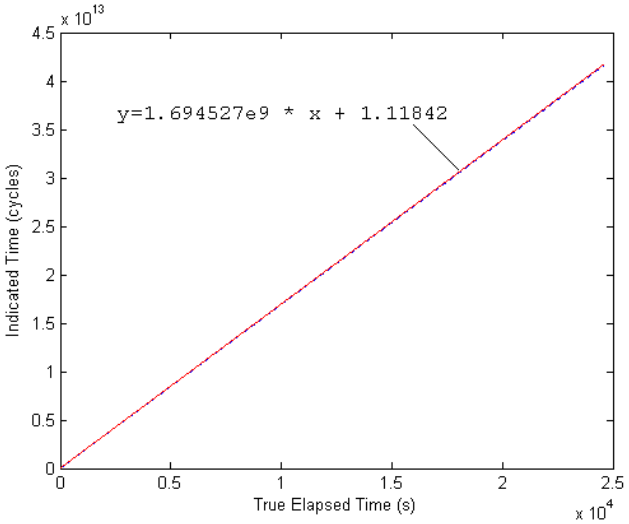

Figure 8 - Accuracy of `rdtsc` timer

More specifically, we observed that the value returned by `rdtsc` instruction increases linearly in time—thereby changes in the value returned by `rdtsc` correspond linearly to actual time elapsed, and provides reliable comparative values of elapsed time. We performed this experiment for all machines on our network, with similar results as presented below.

Figure 8 shows both the indicated time elapsed by the cycle counter and the plot of a curved-fit line, with slope and y-intercept as shown in the figure (here, "True Elapsed Time" is given by the `gettimeofday()` call). We observed in our experiments that measurements of over one hour's duration did not show error in measured time in excess of one half of a standard deviation. We also note that the best-fit slope corresponds precisely to the clocking frequency of the host processor.

5. Experimental Results

We executed each simulated workload with each coherence engine with 1-, 2-, 3-, and 4-machine network configurations. The N=1 configuration is a special case, and uses binaries that do not make calls to our DSM implementation, for efficiency. We normalized all execution times against the runtime of this uniprocessor case. The execution times we examine are taken from the parallel phases of the workloads, ignoring the startup overhead of the DSM system and other initialization times.

As we add additional nodes to the system, they are added in order of decreasing performance. For instance, for the N=2 configuration, the two highest-performance machines participate in the computation. These machines consist of an Intel ® Pentium IV 4.0 GHz-based machine and an Intel ® Celeron 1.70 GHz-based machine. The slower machines are both Pentium III-based, with 800MHz and 600MHz clock speeds.

5.1 Experimental Results for `sea`

From the runtime results for `sea` (Figure 9), we make two major observations. First, as we increase the number of machines operating on the workload, the runtime decreases. This is an expected result. Furthermore, we notice that the coherence engine has a dramatic impact on runtime. Specifically, engines that have the smallest communication overhead perform best for this workload. On all network configurations, we notice that ENGINE_MSI outperforms all other coherence engines. ENGINE_MSI outperforms ENGINE_OU on average by over a factor of 14. We also observe that ENGINE_NC performs almost as well as ENGINE_MSI, but without ensuring consistency. The non-consistent engine suffers a performance penalty for `sea` because it broadcasts on every write to a shared element, and `sea` is write-intensive. A final observation for `sea` is that we obtain speedup over the uniprocessor case for ENGINE_MSI, regardless of the number of nodes. Speedup is not achieved for ENGINE_OU.

The vastly inferior performance of ENGINE_OU can be attributed to its frequent and naïve use of communication with remote nodes to achieve total load/store ordering. Our analysis of executions under this workload show that ENGINE_OU spends 82% of its execution time simply waiting for acknowledgements of coherence messages. ENGINE_NC spends 10% of its time waiting (due to barriers), and ENGINE_MSI spends only 15% waiting.

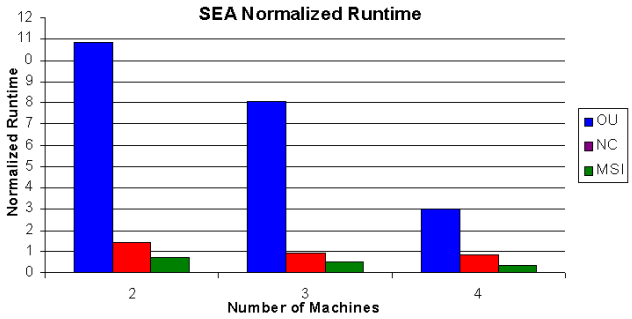

Figure 9 – `sea` Results

5.2 Experimental Results for `genetic`

The results for `genetic` (Figure 10) show some unexpected trends. We notice that an increase in the number of machines participating in computation does not necessarily yield an increase in performance. We attribute this slowdown to the effects of our heterogeneous network. When we add the third and fourth computers (recall that these machines are inferior to the first two), we observe that computation power they contribute does not outweigh the communication overheads they introduce. A similar, more dramatic effect is noted for the third simulated workload. However, as with `sea`, the choice of coherence engine still has a significant effect on the runtime of the workload.

The percent time waiting for responses over the network is 1% for ENGINE_MSI, 9% for ENGINE_NC, and 20% for ENGINE_OU. These numbers are lower for `genetic` than for `sea`, as the number of messages sent is lower relative to computation time.

The only cases for which we obtain speedup with this workload are those using the non-consistent engine with 2 and 3 nodes. Though the `genetic` results were not as

promising as the results for `sea`, they do lend some insight into the behavior of our DSM/IP implementation.

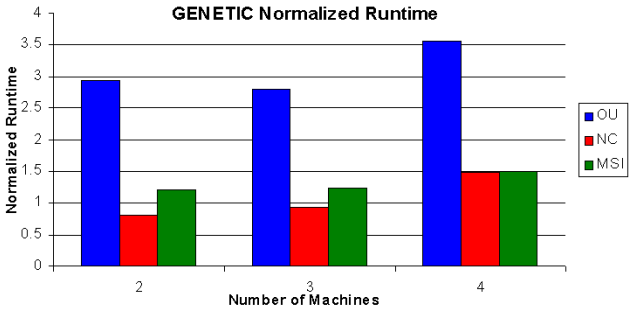

Figure 10 – `genetic` Results

5.3 Experimental Results for `xstate`

From the runtime results for `xstate` (Figure 11), we make several observations. First, we notice speedup over the uniprocessor case for all combinations of engines and network configurations. Next, we see an even more dramatic effect of the heterogeneity of the network. For all coherence engines, as we increase the number of machines in the network, performance degrades. We observe that the average time spent in barriers grows dramatically as we add more nodes. This indicates that the fastest machines finish their computation quickly, then must wait for the slowest machines to finish before proceeding past the barrier.

We also observe that the coherence engine has little effect on the runtime of this workload. In every case, ENGINE_MSI performs slightly better than ENGINE_NC; ENGINE_OU always lags behind the non-consistent engine. However, the difference is less than 3 percent for all cases. This indicates that when execution time is not dependent on communication, the choice of coherence engine is largely irrelevant. This finding is intuitive, as `xstate` is a workload dominated by computation of local, unshared variables, and incurs communication overhead only when a new solution has been found.

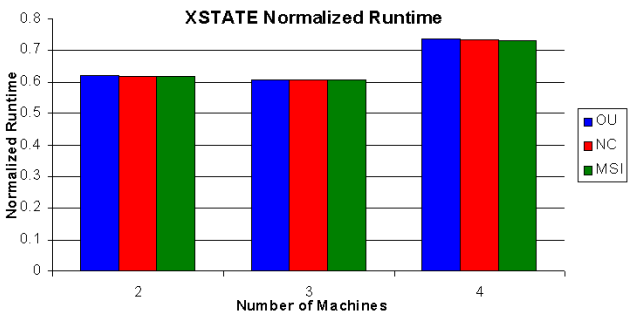

Figure 11 – `xstate` Results

6. Summary

We have presented an implementation of distributed shared memory on clusters of workstations, connected via an IP-based network. Our all-software system consists of a communication backbone, and one of three coherence engines. The CB is responsible for inter-machine communication, and abstracts a reliable network on top of the readily available UDP transport protocol. The coherence engines use the message-passing abstraction provided by the CB to implement shared memory, using three different consistency models and coherence strategies. A simple API for startup and synchronization, built into the CB, is available as black-box primitives for user-level code development.

Evaluation of our system was performed on a small network of four heterogeneous machines, running a commodity operating system. We produced microbenchmarks to test the functional correctness of each coherence engine, and the communication backbone itself. The benchmarks stressed our implementation beyond the normal loads that are generated by DSM-enabled processes, to validate the robustness of our implementation.

For performance evaluation, we produced three simulated workloads, with varying degrees of inter-thread communication and synchronization. These workloads are inspired by true scientific workloads, but are scaled-down for manageability in a short timeframe. The performance of the overall DSM/IP system was measured for all combinations of coherence engine, workload, and network configuration, using cycle-accurate timers and event counters.

7. Concluding Remarks

Let us conclude by first pointing out that, like others before us, we observed application-dependent performance in our DSM/IP implementation. Performance seems to be closely linked to the amount of communication inherent in an application. We observe that sophisticated coherence protocols can partially mitigate the effects of frequent communication. However, communication is extremely latent, and applications requiring a great deal of inter-thread interaction must tolerate this latency under our implementation.

We further observe that the common case may incur too much computation overhead. For each load and store (abstracted as `Read()` and `Write()`), a quick execution time is desirable. The execution time is determined by how many instructions correspond to a `Read()` or `Write()` function call, and whether the current node

has appropriate permissions. If a node does not have sufficient permissions for execution of a request, it must initiate network communication to obtain permissions to read or write (for the consistent engines). In the best case, when a node has sufficient permissions, a single load or store becomes a function call and a check of coherence state, which corresponds to roughly ten x86 assembly instructions, depending on the coherence engine. In the worst case, when we must obtain permissions from other nodes, a single `Read()` or `Write()` can require from one million to 100 million instruction times, depending on the load of the network and the speed of the host processor. We estimate that a single `Read()` or `Write()` can take from 100-1000 instruction times on average, depending on the workload.

Third, our implementation can achieve speedup on small networks, though the improvement is marginal. It would be interesting to examine how well our implementation might perform on larger and/or homogeneous networks.

We also conclude that a heterogeneous network is not ideal for distributed shared memory implementations in which one cannot easily scale work for less efficient computers. If other machines in the network must wait for the slowest to finish its work, this slow computer hurts the aggregate performance of the cluster. One solution to this problem is to assign smaller work sizes to weaker computers. However, this solution is non-trivial, as much information is needed *a priori*, such as which partition of shared memory will be assigned to the slowest computers, and how much work should be assigned to inferior machines. It is possible to perform this division of labor in the workload software, but our implementation cannot guarantee that a particular logical thread will be bound to the same node in each execution.

Despite these challenges, our user-level software implementation of DSM/IP has shown speedup for simulated workloads, even under strict consistency models. Our implementation avoids incurring sharing overhead for unshared local variables, and allows the user to specify an arbitrary sharing granularity, instead of using a fixed, large size. These and other optimizations were instrumental in extracting parallelism from difficult workloads.

References

[1] Sarita V. Adve and Kourosh Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *IEEE Computer*, 29(12):66-76, December 1996.

[2] C. Amza et al., "TreadMarks: Shared Memory Computing on Networks of Workstations," In *IEEE Computer*, 29(2): 18-28, February 1996.

[3] Luiz Andre Barroso, Jeffrey Dean, Urs Holzle. "Web Search For a Planet: The Google Cluster Architecture," In: *IEEE Micro,* 23(2):22-28, March-April 2003.

[4] G. Cabillic and I. Puaut. "Stardust: an environment for parallel programming on networks of heterogeneous workstations." IRISA Technical Report No.1006, April 1996.

[5] J. Carter et al., "Techniques for Reducing Consistency-Related Information in Distributed Shared Memory Systems," *ACM Transactions on Computer Systems*, Vol. 13, No. 3, Aug. 1995, pp. 205-243.

[6] S. Damianakis, A. Bilas, C. Dubnicki, and E. W. Felten. "Client Server Computing on the SHRIMP Multicomputer." In *IEEE Micro*, February 1997.

[7] M. Donahoo, K. Calvert. *TCP/IP Sockets in C: Practical Guide for Programmers*. Morgan Kaufmann Publishers. San Diego, 2001.

[8] B. D. Fleish and G. J. Popek. "Mirage: A Coherent Distributed Shared Memory Design." In: Proceedings of the 14th ACM Symposium on Operating System Principles, 1989.

[9] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, P. Gauthier. "Cluster-Based Scalable Network Services", In: *SOSP-16*, October, 1997.

[10] Mark D. Hill James R. Larus, David A. Wood. "Parallel Computer Research in the Wisconsin Wind Tunnel Project," In: *NSF Conference on Experimental Research in Computer Systems*, Jun. 1996.

[11] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, September 1979, pp. 690-691.

[12] K. Li. "IVY: A Shared Virtual Memory System for Parallel Computing." In *1988 International Conference on Parallel Processing*, 1988.

[13] M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations", In: *Proceedings of the 8th International Conference of Distributed Computing Systems,* pages 104-111, June, 1988.

[14] H. Lu, "Message Passing versus Distributed Shared Memory on Networks of Workstations," Master's thesis, Rice University, Tech. Report Rice Comp-TR-250, ftp cs.rice.edu under public/TreadMarks/papers.

[15] Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/

[16] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token Coherence: Decoupling Performance and Correctness," *International Symposium on Computer Architecture*, June 2003.

[17] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors." In: *ACM Transactions on Computer Systems*. February 1991, pp. 21-65

[18] L. Peterson, B. Davie. *Computer Networks: A Systems Approach.* 3rd Edition. Morgan Kaufmann Publishers. San Diego, 2003.

[19] Steven K. Reinhardt, James R. Larus, David A. Wood. "Typhoon and Tempest: User-level Shared Memory," In: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, April 1994.

[20] M. Schoettner, S. Traub, and P. Schulthess. “A transactional DSM Operating System in Java.” In *Proceedings of the 4th International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, 1998.

[21] A. Silberschatz, P. Galvin, G. Gagne. *Operating System Concepts, 6th Edition.* John Wiley & Sons, Inc. New York, 2002.

[22] E. Speight, H. Abdel-Shafi, and John K. Bennett. “An Integrated Shared-Memory/Message Passing API for Cluster-Based Multicomputing.” In *Proceedings of the Second International Conference on Parallel and Distributed Computing and Networks (PDCN),* December, 1998.

[23] E. Speight and J.K. Bennett. “Brazos: A third generation DSM system.” In *Proceedings of the 1997 USENIX Windows/NT Workshop*, pp. 95-106, August 1997.

[24] M. Stumm and S. Zhou, “Algorithms Implementing Distributed Shared Memory,” *Computer*, Vol. 24, No. 5, May 1990, pp. 54-64.

[25] M. Swanson, L. Stoller, and J. Carter. "Making Distributed Shared Memory Simple, Yet Efficient." In *Proceedings of the Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, March 1998.

[26] P. Sweazey and A. J. Smith, “A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus,” In *Proc. Thirteenth International Symposium on Computer Architecture*, June 1986.

[27] M. Wende, M. Schoettner, O. Schirpf, and P. Schulthess. “Adopting the Internet Protocols in a transactional DSM Operating System.” In *Proceedings of the 4th. World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, 2000.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations” In: *Proc. International Symposium on Computer Architecture*, June 1995.

Signatures

Dan Gibson

Chuck Tsen
