Kernel Level Speculative DSM*

Cristian Ţăpuş†, Justin D. Smith, and Jason Hickey
Caltech Computer Science 256-80, Pasadena, CA 91125
{crt, justins, jyh}@cs.caltech.edu

Abstract

Interprocess communication (IPC) is ubiquitous in today's computing world. One of the simplest mechanisms for IPC is shared memory. We present a system that enhances the System V IPC API to support distributed shared memory (DSM) by using speculations. Speculations provide performance improvements by enabling rollback of overly optimistic speculative executions. This paper describes a speculative total order communication protocol, a speculative sequential consistency model, and a speculative distributed locking mechanism. All these are supported by a mathematical model showing the advantages of speculative execution over traditional execution.

Our DSM system is part of the Mojave system, which consists of a compiler and the extensions of the operating system designed to support speculations and process migration. The goal of our system is to provide a simpler programming paradigm for designers of distributed systems.

1. Introduction

In today's computing world, interprocess communication (IPC) has become an important programming tool. It is no longer conceivable to build significant programs that do not interact with each other, or do not make use of information stored on a remote computer. Several techniques may be used for IPC: message passing, semaphores, shared memory and remote procedure calls. Among these, shared memory provides a method of sharing state and variables between multiple processes that meshes well with most commonly-used programming languages. Designed initially for uniprocessor machines, shared memory was later extended to multiprocessor systems and eventually to distributed systems. Shared memory is appealing because it isolates the programmer from the underlying communication scheme used to maintain consistency across the distributed system.

System V IPC is one of the most widely-used application programmer interfaces (API) for interprocess communication. The API provides a simple way to create and use IPC structures like message queues, semaphores, and shared memory. More recently, a standard POSIX IPC API has been developed that provides many of the same features as System V IPC, albeit with a simplified interface.

Over the last decade, there has been extensive work in the area of distributed systems and shared memory. The concept of distributed shared memory was introduced by Kai Li in 1986. Since then, many systems have emerged. However, due to lack of standardization, most of these systems provide their own API, different enough from each other to require rewrites of the applications when porting them from one system to another.

Our distributed shared memory system is designed as part of Mojave [3], a system consisting of a compiler together with operating system extensions designed to simplify distributed systems programming through the use of high-level language primitives, such as process migration.

Mojave also provides speculations (or speculative execution) to help ensure reliability, introduce fault-tolerance and improve performance. Speculations are used when a process begins a computation based on a condition that may or may not be true. When a process enters a speculation, a copy of its memory and I/O state is created and stored for future reference. If the process later discovers that the condition is false, it rolls back to the point where the speculation was entered by restoring the saved state, undoing all computations that relied on the errant assumption. Once the process verifies that the condition is true, it commits the speculation and discards the saved state.

Speculations improve performance by eliminating blocking for operations that may not immediately complete. One such example is sending messages over the network where the system must normally wait for an acknowledgment that the message was delivered. The Mojave implementation of speculations makes them cheap to enter and commit. When the condition of the speculation is likely to

*This work was supported by the DARPA, grant F33615-98-C3613 and ASCI/ASAP, grant W-7405-ENG-48

†Presenting author.

Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)
0--7695-1919-9/03 $17.00 © 2003 IEEE

occur, it is faster to assume its success than it is to block. Speculations are also beneficial for fault-tolerance, where there is no way to know if there will be a failure until the process evaluates the computation.

This paper presents a distributed shared memory system supporting the System V IPC API. In our system, the compiler and kernel cooperate to provide a simple DSM programming model. In Section 2 we discuss the motivation for our system and continue in Section 3 by presenting the details of the system's design. The current status of our implementation is discussed in Section 4. We provide a comparison of our work with other DSM systems in Section 5 and we conclude by presenting the future directions of our research in Section 6.

Context switch time (in μs)				
	Number of processes			
Process Size	2	5	10	20
size=64k	7.77	46.01	99.90	123.00
size=128k	69.64	196.19	233.09	232.89
size=200k	277.29	281.24	289.84	287.34

Network latencies for remote host (in μs); includes context switch time for two processes
UDP latency using mojave3 (350 bytes): 187.1849
UDP latency using mojave3 (600 bytes): 272.3542
TCP latency using mojave3 (400 bytes): 167.2341
TCP latency using mojave3 (4000 bytes): 342.1603

Figure 1. Benchmarks for Mojave Cluster

2. Motivation

More than a decade before the first distributed shared memory system was implemented, programmers were using the concept of shared memory. System V IPC API was among the first widely accepted libraries that provided users with primitives for shared memory. As the System V API spread across platforms, many programmers developed applications on top of it that made use of the shared memory model. We decided to extend the System V shared memory API to make it suitable for distributed applications. While most of the existing DSM systems provide their own API to distributed shared memory, we believe that using a well known API would make the use of distributed shared memory more appealing to programmers.

DSM systems provide not only different APIs but also different consistency models. Each consistency model gives its own guarantees about when nodes in the system see the changes made by other nodes to the shared memory blocks. Therefore, programmers must adapt their design to meet the requirements of the consistency model. We believe the sequential consistency model (implemented, for instance, in IVY [9]) makes programming with distributed shared memory easier to understand and more portable. One known problem of providing sequential consistency communication latency. Speculations can be used to combat this problem, as we show in the following section.

3. Design

This section will present some of the high level and low level design decisions that we faced along with a mathematical model to support the use of speculations as a way to improve the communication protocol.

One of the main design issues was to determine whether the system should be implemented as a kernel module or a user level application. Most of the existing DSM systems are developed as user level applications that manage memory and provide programmers with a library of functions to be linked with their programs. Our belief was that having the page management mechanism inside the kernel would reduce the overhead of the system.

To ensure consistency of data stored in the shared memory space, our DSM system uses total-order communication among the participants. Total-order protocols are expensive due to the amount of communication required among the machines to achieve consensus on the message order. The latency of messages in the system is also higher because a message cannot be delivered until all other machines have acknowledged receipt, limiting the efficiency to that of the slowest participant. We managed to overcome this problem by using speculations. A model to support our claim is presented in section 3.2.2.

3.1. Kernel level module

In order to determine on which side of the kernel-user boundary our implementation should be, we used a set of benchmarks to measure the context switch time between two processes and the time required to send a large block of data across the network in a Local Area Network (LAN).

The results we obtained from running the LMBench [12] benchmark on our cluster are presented in Figure 1. The cluster consists of 16 machines, each with 700MHz dual processors, running Red Hat 8.0 and the 2.4.18 Linux kernel, and connected through a 100Mbit network. The results indicate that when we have more than 5 actively running processes with size at least 128k, the context switch time is at least as long as sending 400 bytes from one machine to the other.

An interesting observation that we made based on some previously collected benchmarks [13] was that the relative

times spent in context switching and in network latency have changed over the last seven years toward supporting our assumption that context-switching time and network latency are of the same order of magnitude. Our conclusion was that moving the page allocation mechanism inside the kernel could compensate for a consistency algorithm that uses more communication to provide a stricter consistency model.

3.2. Speculations and shared memory

The Mojave system provides speculation primitives that are cheap compared with the time required for the slowest participant to acknowledge delivery in traditional total-order communication systems. Speculations consist of three operations: *entry*, which is performed when a process wants to assume a condition C that may or may not turn out to be true; *commit*, once the process has verified that C is true; and *abort*, which returns the process to a state prior to entry if it discovers that the condition C is false. Multiple speculations may be active at a time; aborting a speculation started at time t implicitly aborts all speculations that were entered after time t. However, committing a speculation started at time t does not automatically commit later speculations.

Our design uses speculation to provide total-order communication with decreased network utilization and message latency. The system uses a weaker causal-order communication protocol which only requires messages that are causally related to be well-ordered. Speculations may also be performed on the DSM operations themselves, to improve fault-tolerance and reduce message latency.

3.2.1 Speculations for total-order messages

Communication among the participants of DSM is causally-ordered using logical timestamps. Each machine is also numbered with a unique node ID. A message is identified by an ID (t, m) that contains both the logical timestamp t and the machine ID m of the sender. Messages are delivered in the order of their logical timestamps; among messages which have the same logical time, the message originating from the machine with smaller ID is delivered first. We introduce a total order relation $<$ on messages as follows: we say message M_1 with ID (t_1, m_1) was issued earlier (or it has a smaller ID) than M_2 with ID (t_2, m_2): $M_1 < M_2$ if $t_1 < t_2$ or $t_1 = t_2 \wedge m_1 < m_2$.

In a total order protocol, when a message M with ID (t_1, m_1) is received, the message is held in a queue until all messages M' with smaller ID have been delivered to the application. We will further use the terms received and delivered as follows: a message is received when the message is passed from the network driver to our protocol, and is delivered to the destination when the protocol makes it available to the application which expects it. To optimize performance, our algorithm uses speculations and a sliding window mechanism, similar to the TCP window. The recipient of a message uses speculation in the following way: if the message has not been delivered within time T, the recipient enters a new speculation that assumes M may be delivered at this time. Once the recipient enters the speculation, it delivers the message. The speculation can be committed once the recipient has seen messages from every other machine with IDs larger than (t_1, m_1). If the recipient, however, receives a message with ID smaller than M, then the recipient must abort the speculation and return to the state where M is waiting to be delivered. Figure 2 gives pseudocode for this algorithm.

```
read message M from network;   t_0 ← time
while time < t_0 + T ∧ ∃ m' . last(m') < M do
    process messages from network
end while
if ∃ m' . last(m') < M then
    enter speculation;   deliver M
    abort if receive message M' s.t. M' < M
    commit when ∀ m' . last(m') > M
else
    deliver M
end if
```

time is current system time
$last(m')$ is last message seen from machine m'

Figure 2. Algorithm for total-order communication using speculations

The window size T should be larger than the round-trip time between the machines. This window size may be maintained dynamically throughout the algorithm using methods similar to those used to maintain the TCP window size. The mathematical model presented in Section 3.2.2 gives formal bounds on the value of T which should be sufficiently large so that the probability of mis-speculation be low, but not too large as we might never get to speculate.

Under this scheme, we no longer require acknowledgments from every machine to agree on the delivery order of messages. In order to keep the speculations' lifetime short, if a machine has not sent a message within an interval $T' > T$, it should send a "quiet" message M_Q to other machines indicating that it has no messages to send. When other machines receive M_Q, they may use the timestamp associated with M_Q to commit speculations for messages with a smaller timestamp.

3.2.2 Mathematical model for speculations

We present a mathematical model for our speculations which will help us better understand the gains and losses of our speculative system. Before we describe our model we would like to introduce the notation used in this section. First, $E[X]$ represents the expected value of random variable X. The notation $E[X]_{low}^{high}$ represents the expected value of random variable X for the range *low-high* of its domain. Thus, $E[X] = E[X]_0^{\infty}$ for a variable X defined on $[0, \infty)$

We want to determine the latency L between the time message M is initially received and the time that M is successfully delivered to the application and compare it to the latency L_c from the non-speculative (classical) model. We will start by showing what the expected latency for the classical model is and then we will describe the specifics of our model and derive the formula for its latency.

For each message M that is received we distinguish two cases: the message could be delivered right away (immediate delivery case), or the message needs to wait until some earlier messages are delivered (delayed delivery case). To represent this distinction we use a parameter p that gives the probability that message M could be delivered right away. We also consider two random variables, U and V which represent, for each case, the time from the reception of the message until the moment we can safely deliver it. Even for the immediate delivery case we might have to wait for a certain period of time until we have the guarantee that it is safe to deliver it. This is the time modeled by random variable U.

The expected latency for the classical model would be given by the following formula:

$$E[L_c] = pE[U] + (1-p)E[V] \tag{1}$$

To complete our model we also have to take into consideration the window size T and the arrival of messages with smaller IDs than the one we are trying to deliver. Let q be the probability that random variable U is greater than the window size T, and r be the probability that random variable V is greater than T. The waiting window size T denotes the time we wait before we start speculating. However, if during time T we can deliver M we do it right away. Let W be a random variable representing the time from the reception of message M until the reception of the last of all messages with IDs smaller than that of M. We know that $W < V$ for the entire domain.

We can proceed to compute the expected latency for our model as follows. Consider the same two cases as for the classical model, described by probability p. For the immediate delivery case, with probability $(1-q)$ the latency is the expected time until we can safely deliver M (since with probability $(1-q)$ the delivery time is less than T) and with probability q the latency is the time we wait until we start speculating plus the time consumed for entering (T_e) and committing (T_c) the speculation. In this case, we know the speculation succeeds without any doubt, so there is no abort time(T_a) involved.

For the delayed delivery case, with probability $(1-r)$ the safe delivery time is less than T. With probability r we will start speculating before delivering the message. However, the time spent speculating until the moment we receive the last message with an ID lower than M's is also part of the latency. When the last message with an ID lower than M is received we would incur the abort time of the current speculation plus the time to enter (T_e) a new speculation, which is guaranteed to succeed, so we also add to it the commit time (T_c).

The exact mathematical formula is given below:

$$\begin{aligned} E[L] = {} & p((1-q)E[U]_0^T + q(T + T_e + T_c)) + \\ & (1-p)((1-r)E[V]_0^T + r(E[W]_T^{\infty} + T_a + T_e + T_c)) \end{aligned} \tag{2}$$

To find the condition for which the latency of our system is less than that of the classical model we require that $E[L] \leq E[L_c]$. This is satisfied if the following inequality holds:

$$\begin{aligned} & pqE[U]_T^{\infty} + (1-p)r(E[V]_T^{\infty} - E[W]_T^{\infty}) \geq \\ & pq(T + T_e + T_c) + (1-p)r(T_a + T_e + T_c) \end{aligned} \tag{3}$$

We used the following formula in rewriting the classical model expected latency:

$$E[X] = p(X < Y)E[X]_{-\infty}^{Y} + (1 - p(X < Y))E[X]_Y^{\infty}$$

Inequality 3 gives us the requirements our system must satisfy to perform better than the classical model. We distinguish again, between the two cases. First, for the immediate delivery case, we improve only if the expected delivery time is greater than the size of the waiting window T plus the time spent to enter and commit the speculation. Second, for the delayed delivery case, we improve only if the time difference between the safe delivery time and the time the last message with lower ID is received is greater than the time spent to abort a speculation plus the time to enter and commit the final speculation.

3.2.3 Speculations on DSM operations

In addition to using speculations to improve the underlying communication for DSM, we can use speculations on the DSM read and write operations to reduce latency and introduce fault tolerance. This section describes a few techniques for applying speculations to DSM operations to further improve performance.

The total-order communication described in the previous section can be adapted to provide a sequential order of the

read and write operations, as follows. A process issuing a read operation for page p would normally have to wait until all writes issued before the read have been processed. In a DSM system, where writes are relatively infrequent, the process can assume that all pending writes for p have been processed as soon as the read request is issued. This speculation is aborted if a write message for p is subsequently delivered for the page with an earlier logical timestamp than the read operation; once the system has verified there are no pending writes for p with earlier logical time, the speculation can be committed. This use of speculations ensures the consistency of the reads while improving performance.

If we additionally want to support fault-tolerance in the event of a machine failure, we also need to add speculations on write calls. If a participant does not receive the message due to failure, the write speculation is aborted and the sender may attempt the write again once it has identified the new set of participants.

3.2.4 Speculations for distributed locking

Speculations can also be used to improve distributed locking mechanisms needed to support synchronization in DSM systems. Shared memory works best when programmers have the ability to use complementary synchronization techniques to avoid race conditions on shared locations. Semaphores can be easily implemented using an optimistic lock acquisition based on speculations. Processes enter a new speculation when the request for the lock is issued; the speculation is committed when the lock is granted and aborted otherwise.

3.3. Sequential Consistency model

The speculative total-order communication and the use of speculations for shared memory accesses and locks allow us to provide the user with a sequential consistency model. The speculations reduce the communication and locking latency and make the memory model more competitive.

The performance problem of sequential consistency was proved by Lipton and Sandberg [10]. They showed that if the read time is r, the write time is w, and the minimal packet transfer time between nodes is t, then $r + w \geq t$. This indicates that by reducing the time spent for one of the operations, the time spent for the other operation must increase. To improve performance our DSM attempts to reduce the value of t by executing optimistically, allowing reduced times for both the read and write operations. If the optimistic assumption fails, performance is not worse than it would have been without using the speculation.

4. Implementation

One of the goals of the implementation was to make it such that it would involve minimal changes in the kernel. We made use of the module system provided by the Linux kernel and we implemented the system as a kernel module for the Linux kernel version 2.4.18. The modifications to the kernel amount to less than 100 lines. The rest of the code is part of a kernel module.

The module starts two pools of kernel threads on the local machine. The *servers* pool serves incoming requests for pages or locks coming from remote machines. The *clients* pool serves local requests to remote machines. When an application accesses a shared page, the page can be either local or at a remote location. In the first case, we serve the access promptly and there is no delay incurred. The system behaves as it normally would for a non-shared memory access. If the page request can not be satisfied locally, we find out the location of the distributed shared memory block and we transfer the page from its remote location. The transfer of data is done at the kernel level directly between clients and servers.

The data structures that we maintain at kernel level are similar to those maintained by the System V implementation. We associate each distributed shared memory block with a distributed key. The main difference is that we provide a global namespace for the distributed keys, using a technique similar to the one used by the domain name system. We have hierarchical DSM nameservers, which ensure the scalability of the system and reply to key inquiries in a timely manner.

Speculations are provided by the Mojave system and are used implicitly by DSM to improve performance of read and write operations. Because speculations are a high-level primitive that is also available to programmers using the Mojave system, DSM must accommodate programs that enter speculations while writing to the shared memory space. To accommodate these *explicit* speculations, our DSM uses techniques that mirror the Mojave system implementation.

The Mojave system uses multiple heaps (one per speculation) to store memory local to a specific process; it also uses a pointer table so that individual pages may be relocated efficiently — all pointers in memory are indices into this pointer table, which contains a pointer to the actual block of memory in the heap. When a process enters a speculation, the current heap is marked read-only. An attempt to write to a read-only page will generate a copy-on-write fault; the page is copied to a new writable heap, and the pointer table is updated to point to the new block. The Mojave system keeps track of the previous block, and in the event that the speculation is aborted, restores the pointer table entry to point to the previous block. When the speculation is committed, the system simply discards the original

Matrix Size	Version of matrix multiplication			
	Par	ShMem	MsgPass	DSM
100	0.006	0.01	0.183	0.08
200	0.046	0.06	0.583	0.34
1000	12.616	11.35	14.106	12.983

Figure 3. Effective computation (in sec)

	Migration type	
Process Size	Source	Binary
size= 100k	2.58	0.72
size= 200k	2.74	0.78
size=1000k	4.30	0.86

Figure 4. Migration time (in sec)

block. With this model, entry and commit are efficient, and only the blocks that are modified must be copied.

Our DSM must provide support in order to undo shared memory writes from a process which rolls back an explicit speculation. This is done by ensuring that the message queue is aware of the speculation, so that any writes done by the process while it is in a speculation are marked as speculative writes. These writes may later be undone using the same mechanism that DSM uses on implicit speculations.

4.1. Experimental Results

We conducted a series of experiments to evaluate the design and performance of our system. The results presented were obtained by running different versions of a matrix multiplication program: a multi-threaded parallel matrix multiplication, a shared memory version using System V IPC, a distributed version using message passing, and the distributed shared memory version compiled with our system.

The implementation of the matrix multiplication programs is the naive one, where for each element in the result matrix we need one row and one column from the multiplying matrices, respectively. We differentiate between the time spent for computation (including memory accesses and communication) and the possible overhead inccured by the use of speculations and process migration, and we analyze them separately.

4.1.1 Effective computation

For each version of the matrix multiplication program we have a main thread that creates the matrices to be multiplied and different threads that do the actual computation of the result matrix. We use square matrices for our tests and we split the result matrix in four equal submatrices, having one thread computing each quarter. The times shown in Figure 3 measure the best time obtained from running the four threads computing the result matrix. For the parallel version of the matrix multiplication program the results shown include the time to start the cloned thread from the main thread, but it does not include the actual time of starting the entire process. All the other times include the time spent in starting the processes.

As it can be seen from Figure 3, our version of DSM always performs better than the message passing version and it is not much worse than the parallel version or the local shared memory version. The main observation that we have to make is that none of the two versions that perform better on a per thread basis can outperform the total running time of the DSM matrix multiplication program because they are bound to run on a single computer. Even when we split the matrix in four on our dual-processor machines, the running time of both parallel and shared memory version is at least twice the running time of one thread. The DSM version incurs a very low overhead when adding migration and speculations, as we will see next.

4.1.2 Migration

Our compiler supports migration as a high-level language primitive. For the purpose of our experiments we timed the migration time for program with different heap sizes. The results are shown in Figure 4. There are two types of migrations: binary migration, when the binary of the program is migrated to its new location, and source migration, when the intermediate representation (IR) of the program is migrated to the new location. The IR migration is more expensive because the program has to be recompiled at destination but it has the advantage of being architecture-independent and it allows the destination machine to verify that the program is type safe and that heap values are used in a proper manner.

For the specific case of the matrix multiplication program, the binary that we migrate has about 200k, so the overhead due to migration is merely 0.7 seconds. Even by adding this to the total time of the DSM version of matrix multiplication, the performance is still better than for the other versions.

4.1.3 Speculations

Speculations are currently not fully suported at the communication protocol level. The results we provide in Figure 5 measure the entry, abort and commit times, for a user level process that uses speculations. The mutation percentile refers to the percentile of the data that changed since we entered the speculation. The abort and commit times are slightly higher than the entry time.

Proc. Size	Operation (and mutation percentile)				
	Entry	Abort		Commit	
		10%	100%	10%	100%
100k	27	65	84	57	54
200k	40	120	135	81	87
1000k	63	131	466	111	109

Figure 5. Speculation overhead (in μsec)

5. Related work

IVY [9] is one of the first software distributed shared memory system. It was developed by Li and Hudak in the late 80s. IVY provides a common virtual address space shared by the machines in the system and supports sequential consistency and a multiple readers-single writer semantics. Sequential consistency provided a programming–friendly model and made the designing of distributed applications easy. However, the system was rather slow due to its page updating mechanism. Our work uses the same consistency model as IVY in the context of the System V IPC API but it provides a faster and more robust updating protocol based on speculations.

The Munin [2, 1] system improved performance through the use of explicit locking and barriers. The resulting weak consistency model decreases communication, but shifts the task of consistency onto the programmer. In the same spirit, Treadmarks [7] further improved the communication overhead but maintained the complexity on the programmer's side. Although the Lazy Release Consistency model (LRC) [6] has some advantages over the previous consistency models, it still relies on a session semantics through explicit use of locking and barriers. We implement a stricter consistency model which provides UNIX semantics and we provide programmers with a simpler programming paradigm. We also manage to reduce the overhead of the communication protocol by using speculations.

One of the more recent DSM systems is DIPC [4]. DIPC is partly implemented in the operating system kernel, but it maintains most of its functionality outside the kernel. It enhances the System V IPC API to include the notion of distributed IPC [5]. In this respect DIPC is similar to our project. However, DIPC's user level daemon is in the critical decision-making path, reducing performance. Another drawback of DIPC is that it has a static "cluster" membership that prevents scalability. DIPC also provides a strict memory consistency through an ownership protocol, decreasing performance.

Gobelins [11] is a Single System Image operating system designed for clusters of PCs which extends the SMP shared memory programming model to clusters. Gobelins introduces special processes and threads which make use of distributed resources (memory and file system) and process migration. Although the system seems very similar to ours, we have a diffferent programming model, based on System V IPC API, and we provide process migration as a programming language feature, supported by our compiler. We also propose a way of improving the performance of the protocol used to maintain sequential consistency by using speculations.

Another area of related work is that of speculative execution systems. There are two main threads of research in this area. One is using general message predictors and pattern-based predictors to learn and predict the memory activity in DSM systems for performance improvement, while the other is using speculative threads to improve computation. The latter direction proposes the use of new architectural designs to allow hardware and software collaboration to support a "monitor and recover" programming paradigm [15].

Cosmos [14] is a general pattern predictor. It accurately predicts the future coherence operations and performs them in a speculative manner to improve performance. By using such a pattern predictor, a predictor-based DSM might eliminate the overhead of maintaining coherence. However, the assumption is that certain branches of programs are iterative and therefore they perform repetitive actions leading to the accurate prediction of the access pattern. Cosmos is able to predict the coherence protocol messages in isolation and has not been integrated with a real coherence protocol. Memory Sharing Predictor (MSPs) [8] is a specialized pattern-based predictor that only predicts memory request messages. MSP eliminates the acknowledgment messages from the pattern tables and reduces this way the overhead and increases prediction accuracy. The MSP sends read-only block copies to predicted requesters and verifies the speculation accuracy based on the write-invalidation messages received from the hosts that did not issue a read before the write.

The main difference between the above systems and our approach is that we are using speculations to provide sequential consistency. In addition, we do not use prediction to determine possible future actions of the system. We use speculations to implement an optimistic protocol that assumes we have seen all the messages sent in the system at the time we enter the speculation.

6. Conclusions

We propose a model for improving performance of a distributed shared memory system by using speculations. We present a speculative total ordering algorithm and a speculative locking mechanism supporting our proposal for a speculative sequential consistency model. The speculative sequential consistency model we illustrated in Section 3 uses

the speculation mechanism provided by the extensions of the Mojave system into the operating system.

As future research, we plan to investigate the use of speculations to develop more efficient communication protocols. We are also interested in applying speculations to distributed file system design in order to provide transparent process migration.

References

[1] J. Bennett, J. Carter, and W. Zwaenepoel. Implementation and performance of munin. In *Proceedings of the 13th Symposium on Operating System Priniciples*, pages 152–164. ACM Press, October 1991.

[2] J. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory using multi-protocol release consistency. In *Dagstuhl Seminar on Operating Systems of the 90s and Beyond*, volume 563 of *Lecture Notes in Computer Science*, pages 56–60. Springer Verlag, 1991.

[3] J. Hickey, J. D. Smith, B. Aydemir, N. Gray, A. Granicz, and C. Ţăpuş. Process migration and transactions using a novel intermediate language. Technical Report caltechCSTR 2002.007, California Institute of Technology, Computer Science, July 2002.

[4] K. Karimi and T. Bynum. dipc sources. http://wallybox.cei.net/dipc/.

[5] K. Karimi and M. Sharifi. Dipc: The linux way of distributed programming. In *The 4th International Linux Conference, Würzburg, Germany*, 1997.

[6] P. Keleher. Lazy release consistency for distributed shared memory. January 1995. Ph.D. Thesis, Rice University.

[7] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, January 1994.

[8] A.-C. Lai and B. Falsafi. Memory sharing predictor: the key to a speculative coherent dsm. In *Proceedings of the 26th annual international symposium on Computer architecture*, pages 172–183. IEEE Computer Society Press, 1999.

[9] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, november 1989.

[10] R. Lipton and J. Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[11] R. Lottiaux and C. Morin. Containers : A sound basis for a true single system image. In *Proceeding of the IEEE International Symposium on Cluster Computing and the Grid*, May 2001.

[12] L. McVoy and C. Staelin. lmbench sources. http://www.bitmover.com/lmbench/.

[13] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. *Usenix*, 1996.

[14] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 179–190. IEEE Press, 1998.

[15] J. Oplinger and M. Lam. Enhancing software reliability using speculative threads. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.