Fault-Tolerant Distributed Shared Memory on a Broadcast-Based Architecture

Constantine Katsinis, *Member*, *IEEE*, and Diana Hecht, *Member*, *IEEE*

Abstract—Due to advances in fiber-optics and VLSI technology, interconnection networks that allow multiple simultaneous broadcasts are becoming feasible. Distributed-shared-memory implementations on such networks promise high performance even for applications with small granularity. This paper presents the architecture of one such implementation, called the Simultaneous Optical Multiprocessor Exchange Bus, and examines the performance of augmented DSM protocols that exploit the natural duplication of data to maintain a recovery memory in each processing node and provide basic fault tolerance. Simulation results show that the additional data duplication necessary to create fault-tolerant DSM causes no reduction in system performance during normal operation and eliminates most of the overhead at checkpoint creation. Under certain conditions, data blocks that are duplicated to maintain the recovery memory are utilized by the underlying DSM protocol, reducing network traffic, and increasing the processor utilization significantly.

Index Terms—Multiprocessors, distributed-shared-memory, fault tolerance.

1 Introduction

Distributed-shared-memory (DSM) is a common programming model supported by multiprocessor systems, along with message passing and data parallel processing. DSM systems offer the application programmer a shared-data model closely related to sequential programming, and reduce the complexity of developing distributed applications.

Popular software-based DSM systems [10], [17], [18] are mostly implemented on top of commercial operating systems running on networks of workstations connected by a switched network and communicating through UDP. Round-trip latencies for small messages tend to be several hundred microseconds, mostly due to software delays. Since the operating system enforces only page-based protection, designers use page-based granularity to prevent the application from writing into a page, and emulate a cache miss through conversion of page faults into messages. Use of large pages as coherence units causes additional network traffic due to false sharing and, as a result, applications exhibiting large granularity show reasonable speedup, while applications with smaller granularity show little improvement. The sequentially consistent memory model is an important factor contributing to the success of DSM, but requires relatively more bandwidth than relaxed consistency models. It is therefore important that interconnection networks be designed with high bisection bandwidth and low latency to provide the best possible performance in DSM systems.

Fault tolerance is a major concern for large DSM systems. As the number of interconnected nodes in the system increases, tolerating node failures becomes essential for parallel applications with large execution times. Backward Error Recovery (BER) [15] allows the system, upon encountering a fault, to restart an application from an earlier, fault-free state. BER requires that the state of a process be periodically saved so that it can be used to restart the application from that last saved state when a node fails.

ICARE [12] is an example of a software-based DSM that makes use of consistent checkpointing in order to create a recoverable DSM system based on an extended write-invalidate coherence protocol that manages both active data and recovery data. ICARE is a software-based DSM using the nodes' memories as large caches, and memory pages as transfer and coherence units.

The Boundary-Restricted coherence protocols are compared in [6], [19], [20] with Write-Invalidate, Write-Invalidate-with-Downgrading, and Write-Broadcast protocols in terms of availability and operating costs. They are designed for a page-based software DSM, and concentrate on maintaining a configurable number of copies of memory pages in the system at any particular time to achieve a certain level of data availability.

A recoverable Release-Consistency software DSM system based on the competitive update protocol is presented in [13]. The goal of the competitive update protocol is to remove copies of pages in nodes that are not actively using them in order to reduce the system overhead of keeping the pages updated. In [3] a software DSM system is presented that relies on lazy release consistency and is extended to support dynamic data replication. It integrates protocol extensions that support fault tolerance with the DSM protocol to reduce overhead.

In [1], a recoverable shared memory (RSM) is proposed in which the memory in the system is divided into two equal parts, the current data blocks and the recovery data blocks. The RSM uses a dependency-tracking matrix in order to create a consistent recovery state, using a two-phase commit protocol, without requiring a global, fully synchronized mechanism for creating a new recovery point. Advantages of this approach are that only processors directly affected by a failure are required to perform a rollback, processor failures

- *C. Katsinis is with the Department of Electrical and Computer Engineering, Drexel University, Philadelphia, PA 19104. E-mail: katsinis@ece.drexel.edu.*
- *D. Hecht is with Rydal Research and Development, Rydal, PA 19046. E-mail: diana@rydalresearch.net.*

Manuscript received 22 July 2003; revised 22 Jan. 2004; accepted 28 Mar. 2004.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0119-0703.

are tolerated transparently, and both sequential consistency and relaxed consistency are supported. However, write operations may be delayed because the copy-on-write mechanism for updating the recovery data requires that the dependency tracking be performed on every write.

DSM systems typically have a large percentage of multicast traffic due to invalidation messages in Write-Invalidate cache coherence protocols and update messages in Write-Update protocols. There is a large amount of research in multicast communications in popular switch-based architectures with path-based broadcasting [9], trees, and wormhole routing [2]. Large efforts are focused on development of extensive algorithms to alleviate the fact that intense multicast communications cause wormhole routing to resemble store-and-forward routing. Broadcast-based networks are an alternative to current switch-based networks. They offer nonblocking communication, high bandwidth (scaling directly with the number of nodes) and low latency. Since nodes appear the same from any point of view, the user (or the compiler) is free to structure the data and the application code to reflect the inherent parallelism. Specific implementations can be constructed using optoelectronic devices, relying on sources, modulators and arrays of detectors, all being coupled to local electronic processors.

This paper examines an implementation of a broadcast-based architecture and presents extensions to the traditional DSM protocol, such that the resulting system may tolerate a node failure. Network activities relating to fault-tolerance and cache coherence are fully exploited to the extent that processor utilization (during normal operation) is increased and the time used to create checkpoints is decreased, further contributing to execution speedup. The augmented protocol is compared with a simpler protocol that performs DSM and FT activities separately, using real application traces and synthetic loads executing on the broadcast architecture. Section 2 presents the broadcast-based architecture, Section 3 examines the operation of DSM and the design of the cache and directory controllers, Section 4 presents the fault-tolerant DSM protocol, Section 5 presents a queuing-network model of the architecture, Section 6 presents the simulator and the workloads, and Section 7 compares the performance of the fault-tolerant DSM protocols.

2 The SOME-Bus Architecture

One implementation of an architecture that can support simultaneous multicasts from all nodes is the Simultaneous Optical Multiprocessor Exchange Bus (SOME-Bus) which has been presented in [11] and is summarized here. It is a low-latency, high-bandwidth, fiber-optic network that directly connects each node to all other nodes without contention. In general, the SOME-Bus contains N nodes and N dedicated broadcast channels (each with W wavelengths). The channels are implemented by K fibers, each carrying M wavelengths. Each of the N nodes also has an input channel interface based on an array of N receivers (each with W detectors) that simultaneously monitors all N channels. The total number of fibers is $K = NW/M$. A simple configuration with $N = 128$ nodes and $W = 1$ wavelength per channel requires $K = 32$ fibers with $M = 4$ wavelengths per fiber, and a receiver array at each node containing 128 detectors organized as 32×4 over the surface of a single chip [16]. Each node uses separate micromirrors [14] and laser sources to insert each wavelength of its channel. Slant Bragg gratings [4] written directly into the fiber core are used as narrow-band, inexpensive output couplers. This coupling of the evanescent field allows the traffic to continue and eliminates the need for regeneration. Photodetectors are created through a layer of amorphous silicon (a-Si) on the surface of electronic processing devices. Due to the low conductivity of the a-Si layer, no subsequent patterning is required and, therefore, the yield and cost of the receiver is determined by the yield and cost of the CMOS device itself. Optical power budget analysis of a system with 128 nodes, 32 wavelengths per fiber, and 10 mW of power inserted into the fiber shows that the worst case for output power, occurring where light from the first node is coupled out by the receiver at the last node, is 21microW, which is sufficient for present detectors.

The receiver array does not perform any routing and, consequently, its hardware complexity is small. It contains an optical interface that performs address filtering, barrier processing, length monitoring, and type decoding. If a valid address is detected in the message header, the message is placed in a queue, otherwise, the message is ignored. The address filter can recognize multicast group addresses as well as broadcast addresses in addition to recognizing the address of the host node. The receiver array also contains a set of queues such that one queue is associated with each input channel, allowing messages from any number of nodes to arrive and be buffered simultaneously. This organization supports multiple simultaneous broadcasts, provides bandwidth that scales directly with the number of nodes in the system, and eliminates the need for global arbitration. Arbitration may be required only locally in the receiver array when multiple input queues contain messages. In the current design, they are served round-robin.

3 DSM on the SOME-Bus

A natural implementation of a DSM interconnected by the SOME-Bus architecture is a multiprocessor consisting of a set of nodes organized as a CC-NUMA system [8]. Each node contains a processor with cache, memory, directory, output channel, and a receiver array. The SOME-Bus provides an advantage over software-based cache coherence approaches because it supports a strong integration of the transmitter, receiver, and cache controller hardware producing an effective system-wide hardware-based cache coherence mechanism [8]. Maintaining coherence is not dependent on the operating system and, therefore, the large latencies due to the underlying message-passing protocol stack are not encountered.

A sequential-consistency model is used with statically distributed directories enforcing a Write-Invalidate protocol. The typical MSI protocol is used, with the directory controller multicasting INV-REQ and DNGR-REQ messages to the relevant nodes. (Fig. 1 shows all message types.) In the SOME-Bus, messages are physically broadcast over the sending node's output channel. The receiver address-matching logic of a particular node examines the header of the incoming message to determine if it is intended for that node, and rejects the message if necessary by not placing it in the input queue. The broadcast becomes effectively a multicast, and the decision to accept or reject an input message is performed at the receiver input rather than the cache controller of each remote node. Consequently, the cache controller in every node receives only the

DATA-REQ	Data request
DATA-ACK	Data acknowledge
OWNR-REQ	Ownership request
OWNR-ACK	Ownership acknowledge
DNGR-REQ	Downgrade request
DNGR-ACK-WB	Downgrade acknowledge with writeback
INV-REQ	Invalidation request
INV-ACK	Invalidation acknowledge
INV-ACK-WB	Invalidation acknowledge with writeback
VICTIM-WB	Victim writeback

Fig. 1. Message types.

messages that pertain to data blocks resident in that node's cache. The channel controller receives messages from the cache or directory controllers and delivers them to the destination node. If the source and destination nodes of the message are different, the message is considered to be remote and is placed on the output queue associated with the output channel of the source node. If the source and destination node of a message are the same, the message is considered to be local and the channel controller places the message directly on the input queue of the controller (directory or cache) to which the message is directed.

To implement DSM as efficiently as possible on the SOME-Bus architecture, the channel controller is closely integrated with the cache and the directory. This organization offers the fastest access to data in the cache and the directory and results in smaller latencies. Fig. 2 shows the major blocks of the channel controller as well as the relating functionality of the cache and the directory. A processor reference may result in a cache miss and a subsequent interaction with the directory. If necessary, a request is created by the directory and is forwarded to the channel controller, which creates a request message and enqueues it in the output channel queue for transmission. The channel receiver passes incoming messages to either the cache or the directory.

To maintain high performance, the receiver can simultaneously extract two messages (directed to the cache and the directory) from distinct channel input queues. The dual resolver is essentially a multiplexer that selects the next active queue in round-robin and forwards its index and data output to the proper data path. When necessary, the resolver can be set to select a specific queue. If both resolvers decide to read the same queue, the cache controller is given priority to avoid having the processor wait for the arrival of DATA-ACK or OWNR-ACK messages. To avoid Head-of-the-Line blocking, input queues are organized as linked lists, allowing messages to be removed from anywhere in the queue. Separate pointers for the cache controller and the directory controller are used to obtain the next message to be examined (or removed) by each controller.

4 Fault Tolerance

A multiprocessor system based on the SOME-Bus can be composed of hundreds of nodes. As the number of nodes in the system increases, the probability that the system will experience temporary faults due to node failures also increases. For this reason, the ability to tolerate node failures becomes essential for parallel applications with large execution times.

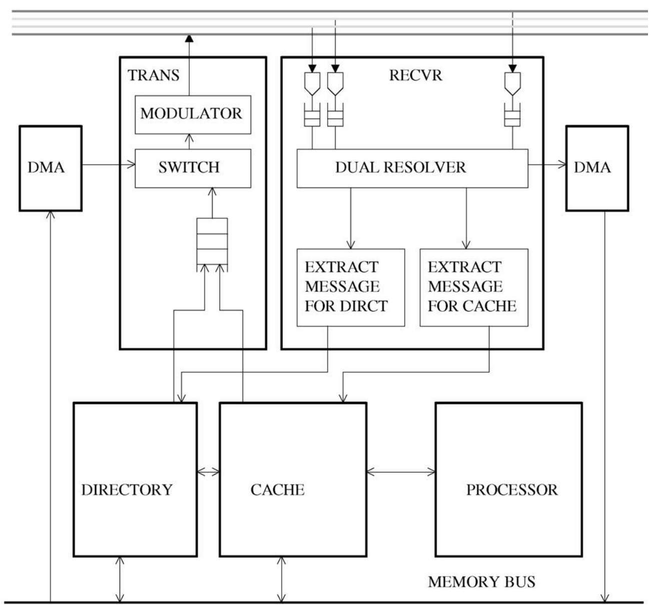

Fig. 2. Controller organization.

Backward Error Recovery (BER) techniques are based on checkpointing. Coordinated checkpointing requires processes to synchronize before creating a new checkpoint in order to provide a consistent global state. The goal in coordinated checkpointing is to limit the rollback to the last saved checkpoint. The approach only requires a single checkpoint per processor to be saved. Uncoordinated checkpointing approaches allow processors to save their state independently from the other processors. Although it avoids the synchronization overhead, it is susceptible to rollback propagation where a domino effect can cause the rollback to extend back to the initial state of the program. Log-based methods are aimed at eliminating the domino effect while still supporting uncoordinated checkpointing. A checkpoint is periodically created (independently) for each processor and incoming messages are logged in the interval between checkpoint creation. At recovery, the previous checkpoint is recovered and the logged messages that occurred before the failure are replayed to achieve a consistent state.

Pessimistic log-based methods require that each received message be logged into stable storage before it can be processed and, consequently, result in a fault-free runtime cost due to the extended processing time for each message. Optimistic log-based methods allow for the asynchronous logging of messages in volatile storage and then periodically flush the messages to stable storage. While this approach reduces the fault-free runtime cost associated with pessimistic logging, it requires complicated recovery algorithms and garbage collection mechanisms. In addition, optimistic logging can result in an unbounded rollback problem caused by messages whose logs are lost before being logged to stable storage. More than one checkpoint is required for this approach to handle the rollback propagation problem. Causal logging approaches have a low failure free overhead and limit the rollback to the most recent checkpoint, but require complex causality tracking and recovery algorithms.

The SOME-Bus architecture takes advantage of Backward Error Recovery (BER) techniques by relying on consistent checkpointing because of its advantages in terms of low failure-free cost, a limited rollback effect and simple checkpointing and recovery algorithms. The costs associated with synchronizing before taking a checkpoint are minimized due to the ability of the SOME-bus to support multiple simultaneous broadcasts which allows for the efficient support of high-speed, distributed barrier synchronization mechanisms. In all of the checkpoint or log-based schemes described above, at least one copy of the recovery memory is saved per node. The SOME-Bus requires that two checkpoints are saved per node (one for itself and one saved as an independent backup for another node). Under the FT1 protocol (described below), a copy of the local memory of another node is also saved.

Although it is possible to save checkpoints to disk, the low bandwidth and high latency of these devices can adversely affect the time required to create a checkpoint, especially if each processing node does not have its own local disk. In order to reduce the time required to create checkpoints, saving checkpoints in the volatile memory of remote nodes satisfies the requirement for stable storage due to the replication of the data in two distinct sites [5], [12]. This approach requires additional memory but offers much higher bandwidth than disk-based memory and is scalable since adding more processors results in more memory. It can be extended to tolerate multiple node failures when all copies of the checkpoint do not reside on the failed nodes. If memory is limited and local disks are available at each node, it is possible to store the Recovery Data on disk. The Home2 copy of Local Memory used in the FT1 protocol (described below) is not placed on disk however because it is accessed to fill read requests at runtime.

A system based on the SOME-Bus architecture can exploit the natural broadcast of messages and can use the DSM mechanisms for data replication to hide some of the overhead associated with fault tolerance. In addition, it can use the data maintained for fault tolerance to improve the DSM performance during normal operation. In the following, two protocols are examined: Protocol FT0 uses separate DSM and FT activities to maintain cache coherence and fault tolerance, and protocol FT1 relies on message broadcasts to combine DSM and FT activities, hide most of the cost due to fault tolerance, and improve system performance. The issues of protocol FT0 have been examined in the past; it is summarized here because it can also take advantage of the SOME-Bus architecture and to form a basis of comparison for the FT1 protocol. Since synchronization is relatively inexpensive in the SOME-Bus architecture, the following discussion assumes that all processors synchronize before a checkpoint is taken. Also, it is assumed that the network is reliable and that processors are fail-stop; they either work correctly, or crash without corrupting data. When a processor crashes, the contents of its memory are lost.

FT0 Protocol: In the FT0 protocol, each node keeps a full copy of its local memory, called the recovery memory. In addition, every node also contains a copy of the recovery memory of another node. The memory layout is determined statically. The total address space is divided into equal sections and each processor has one section of the total address space as well as the corresponding directory structures. A node that locally contains and manages a section of the global memory space is referred to the Home node for those memory blocks. For simplicity, we assume that the Home node for a particular data block can be found by dividing the block address by the total number of blocks per node.

If Node Y encounters an application error that does not contaminate its recovery memory, Node Y can use its own recovery memory to restore the previous checkpoint state and then restart along with the rest of the nodes in the system. In the case of a complete failure and replacement of Node Y, the copy of Node Y's recovery data that resides on Node X can be used to initialize the replacement node. In either case, after the memory of Node Y has been restored, the system can perform Backward Error Recovery.

Creating a checkpoint includes a cache-writeback phase, where all modified blocks are written back to their corresponding home nodes, and a recovery-memory update phase, where the directory controller copies the blocks in local memory that have been modified since the previous checkpoint to the recovery memory. At the same time, these modified blocks are added to an update message that is sent to the backup node so that it may update the backup copy of the recovery memory. A fault that occurs during this phase could cause the recovery memory to become corrupted. For this reason, the recovery memory update operation must be atomic. A temporary copy of the recovery memory is created with the required updates. When this copy has been updated successfully on every node, the processors synchronize again and the newly created copy of the recovery memory becomes the recovery memory for the current checkpoint and the previous recovery memory is deleted. All nodes synchronize again and then resume normal operation. If any node has a failure while the new copy of the recovery data is being created, the checkpointing procedure can be stopped and then restarted using the original recovery memory that remained unmodified. The original recovery memory is not deleted until every node indicates that it has successfully created the new recovery memory.

The FT0 protocol does not cause any additional network traffic associated with fault-tolerance during normal execution. The overhead occurs only when a checkpoint is being created and is caused by the messages related to the cache-writeback phase and the exchange of update messages between each node and its backup node during the recovery-memory-update phase.

FT1 Protocol: Simulation results show that the largest source of overhead from checkpointing under FT0 is the amount of time required to transfer the updates for the recovery memory to the backup node. The FT1 Protocol significantly reduces this overhead by eliminating the update message exchange during the recovery-memory-update phase. Instead, the backup node keeps a consistent backup copy of the corresponding primary node's local memory in addition to the backup copy of the primary node's recovery memory. During the recovery-memory-update phase of FT1, the backup copy of the recovery memory is updated from the corresponding backup copy of local memory, requiring no additional network communication.

Fig. 3 shows the organization of a node's memory for the FT1 Protocol. Node X contains its own local memory and recovery memory as well as a copy of Node Y's local memory and Node Y's recovery memory. Node X is the

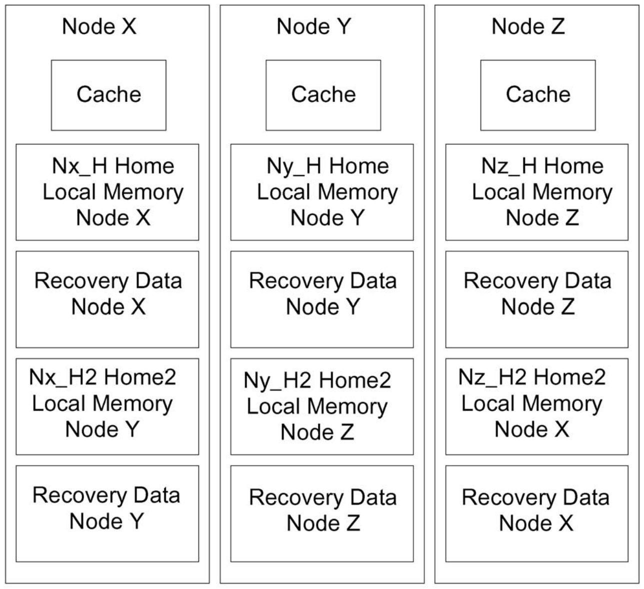
Node X
Cache
Nx_H Home Local Memory Node X
Recovery Data Node X
Nx_H2 Home2 Local Memory Node Y
Recovery Data Node Y
Node Y
Cache
Ny_H Home Local Memory Node Y
Recovery Data Node Y
Ny_H2 Home2 Local Memory Node Z
Recovery Data Node Z
Node Z
Cache
Nz_H Home Local Memory Node Z
Recovery Data Node Z
Nz_H2 Home2 Local Memory Node X
Recovery Data Node X

Fig. 3. FT1 node memory organization.

Home node for data blocks located in its own local memory (Nx_H). In addition, Node X is the Home2 node for data blocks located in the copy of Node Y's local memory which resides on Node X (Nx_H2). Similarly, Node Y contains Home memory (Ny_H) and Home2 memory (Ny_H2) for blocks located in the copy of Node Z's local memory which resides on Node Y.

The checkpointing procedure for FT1 is as follows: Assume Node X's cache contains a data block whose home is located on Node Z. During the cache writeback phase, the cache controller at Node X multicasts the writeback to both Node Z, update the home local memory, and Node Y, to update the Home2 local memory. During the recovery memory update phase, both sets of recovery data are updated from the corresponding local memory with no network communication. For example, Node Y updates the recovery data for Node Y from the Home local memory and updates the recovery data for Node Z from the Home2 local memory.

Keeping the checkpoints in memory, as opposed to writing the checkpoints to disk, increases the performance. However, if the memory is limited and stable disk storage is local to each node, both sets of recovery data may be written to disk at each checkpoint. The Home2 local memory data, used in FT1 to fill read requests, increases the runtime DSM functionality and improves performance and, therefore, should remain in memory and not be written to disk. This approach limits the total memory requirements under FT1to at most twice the amount of local memory required when no fault tolerance is used.

Recovery after a node failure involves the replacement of the failed node by a spare node on the SOME-Bus and the transfer of recovery memory contents into the new node. Processors are assumed fail-stop; they either work correctly or crash without corrupting data. We examine three nodes in the system X, Y, and Z such that X is the backup node for Y and Y is the backup node for Z. If Node Y has a permanent failure, it is essentially removed from the architecture by disabling its network controller, and a replacement Node A is enabled and assigned the network ID of the failed Node Y. Node A receives Node Y's recovery data from Node X and also receives the backup recovery data for Node Z from Node Z (the backup recovery data for Node Z had been stored on Node Y and has now been lost). No additional changes to any data structures such as directories or routing tables are needed. Since the replacement Node A is assigned the ID of the failed Node Y, it will begin receiving messages that are directed to Node Y.

The time involved in the recovery is the time it takes to synchronize the processors, send two messages containing Recovery Data to the replacement node, signal all processors to roll back to the previous checkpoint (update their local memory with the local copy of their recovery data) and synchronize the processors and continue execution.

Home2 memory access: The data replication that occurs as a result of keeping the Home2 local memory consistent is used to enhance the performance of the DSM system by allowing a Data request to be filled from either the Home or Home2 memory. Network traffic and service latency for the data request is reduced since the request is filled from the memory on the same node rather than from memory on a remote node. As long as a block in Nx_H2 is in the SHARED state, it can be used to fill a read request from a process at Node X locally.

In the FT1 protocol, a Node X takes advantage of the SOME-Bus capabilities and keeps its backup copy of Node Y's local memory (Nx_H2) consistent by receiving a copy of any cache writebacks to Node Y as they occur. This approach ensures that backup Node X has all the information necessary to update its copy of Node Y's backup recovery memory without relying on the home node (Node Y) to explicitly send it at checkpoint time. Keeping the Home2 local memory consistent requires that when ownership of a block is requested, the Home node must send an INV-REQ message to the Home2 node in addition to any other nodes sharing the block. In the SOME-Bus architecture, a single INV-REQ is multicast to all nodes that have a copy of the block. The effect of this additional message depends on whether the relevant block is shared by other nodes or only by the Home2 node. In the first case, the time spent collecting an additional INV-ACK messages from Home2 is relatively small and its effect diminishes as the number of sharers increases [8]. In the second case, where no other INV-ACKs need to be collected by other nodes, the entire time spent sending the INV-REQ to Home2 and collecting the INV-ACK directly adds to the latency of the ownership request. As a result, the FT1 protocol incurs a penalty in the form of increased network traffic, while the FT0 protocol does not incur any such penalty. However, two properties of the FT1 protocol tend to reduce network traffic: 1) creating a checkpoint requires significantly fewer messages and 2) since memory Nx_H2 remains consistent, processes in Node X may access it directly without having to send or receive messages from node Y.

Maintaining Home2 memory consistency: Home2 must receive a copy of all coherence-related messages so that it may determine the order in which the DATA-ACK and OWNR-ACK messages were sent by Home. In the SOME-Bus architecture, each node has a separate input queue for every channel in the system. While messages in each specific input queue are removed in-order, input queues may be selected at random. Consequently, if several input queues of one node contain writeback messages (originating from different nodes) for a particular data block, the proper processing order of these messages cannot be determined simply by their queue positions. This problem is resolved by requiring that the Home node multicast

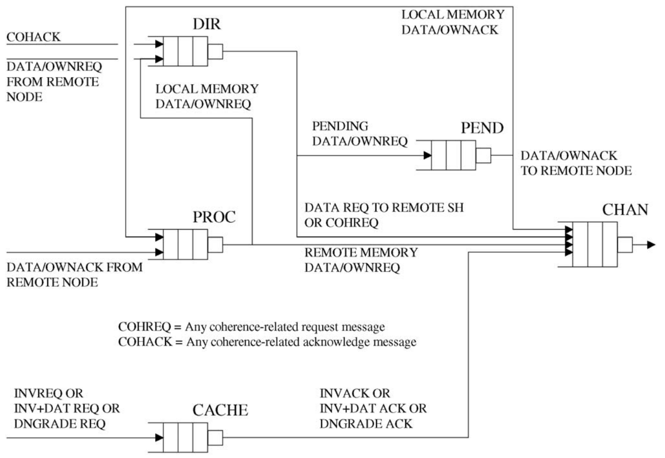
COHACK
DATA/OWNREQ FROM REMOTE NODE
DIR
LOCAL MEMORY DATA/OWNACK
LOCAL MEMORY DATA/OWNREQ
PENDING DATA/OWNREQ
PEND
DATA/OWNACK TO REMOTE NODE
PROC
DATA REQ TO REMOTE SH OR COHREQ
CHAN
REMOTE MEMORY DATA/OWNREQ
DATA/OWNACK FROM REMOTE NODE
COHREQ = Any coherence-related request message
COHACK = Any coherence-related acknowledge message
INVREQ OR INV+DAT REQ OR DNGRADE REQ
CACHE
INVACK OR INV+DAT ACK OR DNGRADE ACK

Fig. 4. Node queues.

DATA-ACK and OWNR-ACK messages to the requesting node and to the Home2 node. In addition, writeback messages are multicast to Home and Home2 nodes.

Serialization of the write access to a data block is enforced at the Home node. Since all messages sent by the Home node are received by all nodes in the same order, the Home2 node can determine the correct sequence of nodes that obtained ownership for a particular data block by searching for OWNR-ACK messages for the data block in the Home2 input queue that receives messages from the Home node. Home2 then matches writeback messages received from any node with corresponding Acknowledge messages received from Home.

When Home2 finds an OWNR-ACK message for a data block b in the queue associated with Home, it searches the queue in FIFO order to locate the next DATA-ACK or OWNR-ACK message for data block b. If a DATA-ACK message follows an OWNR-ACK message directed to node X, Home2 searches the input queue associated with node X in FIFO order for a DNGR-ACK-WB message or a Victim-WB message for data block b. The data block in the Home2 memory is updated with the writeback value and is set to the SHARED state. All three messages (DATA-ACK, OWNR-ACK, and DNGR-ACK-WB or Victim-WB) are removed from the input queues.

If Home2 finds an OWNR-ACK message directed to node Y following an OWNR-ACK message directed to node X, it searches the input queue associated with node X in FIFO order for an INV-ACK-WB message or a Victim-WB message. In either of these cases, the data block is not changed since it will remain in the INVALID state. The OWNR-ACK message directed to node X and the writeback message are removed from the input queues. The OWNR-ACK message directed to node Y is not removed so that it may be matched with subsequent Acknowledge messages.

5 Queuing Network Model

The system consisting of the processor, the cache, the directory and the channel controller can be represented by a set of queues through which messages of different types flow. Fig. 4 shows the queue organization. Although no messages are delivered directly to the processor, the arrival of DATA-ACK and OWNR-ACK messages causes threads

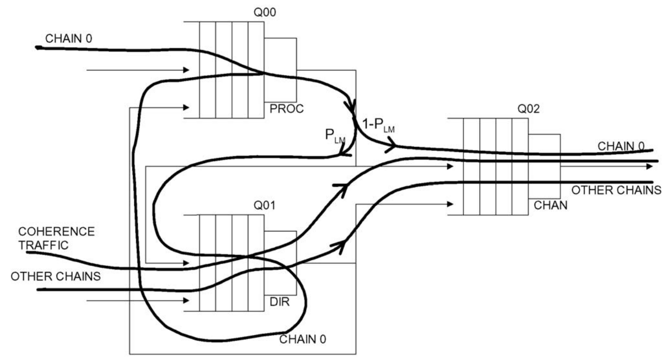

Fig. 5. Traffic through one node (Node 0).

to become ready for execution and, therefore, affects the processor operation. Messages of other types used to support coherence arrive at either the directory or the cache queue.

Let M be the number of parallel threads per node. Fig. 6 shows all symbols used in the analysis of this section. When a node has one or more active threads, it generates messages with mean interval of R time units between requests. A request message generated by a node is directed to any other node with equal probability [7]. This operation can be represented by a multichain, closed queuing network, where messages receive complicated forms of service at certain server stations. The M threads owned by each node form a separate class of messages with population equal to M. When $m < M$ threads have outstanding messages, the remaining M-m messages are served in FCFS order at the processor.

Due to the symmetry of the system, all directory controllers behave in a similar fashion. Therefore, a system with $N = M + 1$ nodes shows the same performance as a system with $N > M + 1$ nodes. Typically, a small number of threads per node is possible. Then, a relatively small queuing network representing a SOME-Bus system with $M + 1$ nodes can be used to calculate performance measures. Node $P (P = 0, 1, \ldots, M)$ owns M regular data messages that circulate through the processor, the directory controllers of node P, and the other M nodes and the necessary channels. Different chains are used to distinguish between messages belonging to

K_D	total number of data messages in the system
K_C	mean number of coherence messages in the system
a_D	arrival rate of coherence requests
s_d	directory controller response time (mean S_d)
L_1	time to compose a data acknowledge message
t_c	coherence message channel transfer time (mean T_c)
w_{ch}	mean channel queue time (mean W_{ch})
w_{ca}	mean cache queue time (mean W_{ca})
t_{inv}	time to send INV-REQ and collect INV-ACKs (mean T_{inv})
Y_{inv}	mean time to collect INV-ACKs
M	number of parallel threads per node
N	number of nodes and number of channels
N_{inv}	number of blocks to be invalidated on ownership request
p_{rd}	probability data message is due to read miss
p_{sh}	probability block is found in shared state
R	mean thread runtime
R_D	mean data message roundtrip time
s_{ca}	mean cache response time (mean S_{ca})
t_d	mean data message channel transfer time

Fig. 6. List of symbols.

different nodes, so that messages in chain P belong to node P. Messages owned by node P are sent to the directory of any node, including the owner.

The complexity of the model arises from the fact that there are several other messages that are used only to maintain coherence. These messages are generated by the directory controllers, go through the channels, possibly interact with cache controllers, and return to the originating directory controllers. Fig. 5 shows the traffic through one node. This additional coherence traffic has two direct effects on the regular data messages of the chains.

First, the coherence traffic determines the service time at the directory controller. There are four distinct cases: DATA-REQ to a block in SHARED state (prob. $p_{rd}p_{sh}$), DATA-REQ to a block in EXCLUSIVE state (prob. $p_{rd}(1-p_{sh})$), OWNR-REQ to a block in EXCLUSIVE state (prob. $(1-p_{rd})(1-p_{sh})$), and OWNR-REQ to a block in SHARED state (prob. $(1-p_{rd})p_{sh}$). The mean service time experienced by a data message at the home directory is therefore

$$L = L_1 p_{rd} p_{sh} + 2(T_c + W_{ch})(p_{rd}(1-p_{sh}) + (1-p_{rd})(1-p_{sh})) + T_{inv}(1-p_{rd})p_{sh},$$

where L_1 is the time that the directory needs to compose the DATA-ACK message with the copy of the requested block, T_c is the mean coherence message channel transfer time, W_{ch} is the mean channel queue time, and $T_{inv} = (T_c + W_{ch}) + Y_{inv}$, where T_{inv} is the mean time to send the INV-REQ and collect the INV-ACK messages. If the number of blocks which must be invalidated on each ownership request is N_{inv}, then Y_{inv} is the mean value of a random variable equal to the maximum of N_{inv} identical random variables, each being equal to the service and queuing time at the channel server. We assume that the cache controller works at much higher speed than the network, so that the response time of the cache controller can be ignored.

Second, the coherence traffic is also passing through the channels and absorbs a fraction of the channel service rate, making the channel service time of the data messages appear larger. The apparent service time is $T' = (T * (K_D + b * K_C))/K_D$, where T is the real channel mean transfer time, $K_D = M(M+1)$ is the total number of data messages in the system, K_C is the mean number of coherence traffic messages present in the system, and b is the ratio of the mean coherence message size to the mean data message size. Since coherence messages are created due to the arrival of data messages into the underlying coherence mechanism, we use Little's result, $K_C = a_D t_{inv}$, where a_D is the arrival rate of data messages into the underlying coherence mechanism and t_{inv} is the lifetime of a coherence message. The arrival rate a_D is $a_D = (1-p_{rd}p_{sh}) K_D/R_D$, where R_D is the data message mean roundtrip time, and the $(1-p_{rd}p_{sh})$ factor is necessary because DATA-REQ messages to a SHARED block do not result in coherence traffic.

Assuming geometrically distributed processing times, this queuing network model is in a condition that may be solved by standard techniques, with the only exception that the service time at the directory server depends on the queuing time at the channel server queue. Simulation results show a moderate channel utilization which results in relatively small queuing times. To keep the queuing network model simple, we ignore the dependence of the directory service time L on the channel wait time W_{ch}. Using the reduced channel service rate, and the estimate of the directory service time L, the closed queuing network with $3(M+1)$ queues, $(M+1)$ chains, and M messages in each chain can be solved using standard closed-network techniques.

To perform a comparison between simulation and theoretical results, the parameters of the queuing network model must be related to key properties of the application program. The inputs to the theoretical model are the mean service times at the processor, directory and channel servers, and the probability that a miss can be satisfied by a block in the local node memory. The directory and channel service times are calculated using the equations shown above. The processor mean service time and the local access probability are parameters that characterize the application behavior. The relevant probabilities are P_{Read} (memory read access), P_{Miss} (cache miss), P_{Loca} (local access), P_{Modi} (prob. that node finds one of its blocks modified by another node), P_{Upgr} (prob. that write access finds the required block in cache in SHARED state), and p_{Shm} (prob. that node finds one of its blocks SHARED or EXCLUSIVE). Parameter P_{Loca} indicates the degree that individual processes in the application tend to access the local memory of the node that they run on. Similarly, P_{modi} and P_{Upgr} indicate the degree of interaction between processes as they read and write in shared areas of the distributed memory. Parameter P_{Shm} also relates to process locality and indicates the degree that blocks tend to be accessed repeatedly and, consequently, are found in SHARED or EXCLUSIVE state and not UNOWNED state.

From these parameters, we calculate the probabilities P_{ml} and P_{mr} that a memory reference would result in a request to the local directory or a remote directory, respectively,

$$\begin{aligned} P_{ml} &= P_{read}P_{Miss}P_{Loca}P_{modi} + (1-P_{Read})P_{Loca} \\ &\quad (P_{Hit}P_{Upgr} + P_{Miss}P_{Shm}) \\ P_{mr} &= P_{Read}P_{Miss}P_{Remo} + (1-P_{Read})(1-P_{Loca}) \\ &\quad (P_{Hit}P_{Upgr} + P_{Miss}), \end{aligned}$$

where $P_{Hit} = 1 - P_{miss}$ is the probability of a cache hit. Then, the probability that a request message is sent to the local directory is $P_{LM} = P_{ml}/(P_{ml} + P_{mr})$, and the mean processor runtime before a message is generated is $R = 1/(P_{ml} + P_{mr})$. These parameters complete the specification of the model shown in Fig. 5, which can be solved to produce server utilizations and queue waiting times.

6 SIMULATION

The performance of the SOME-Bus architecture using the FT0 and FT1 protocols is examined using a simulator that maintains the precise state of the processor, directory controller, cache controller, and channel controller, for every memory reference and every message. The applications executed by the simulator consist of sequences of memory read or write references extracted from multiprocessor address trace files or artificially generated address streams.

A set of three multiprocessor address trace files for programs called *speech*, *simple*, and *weather* was obtained from the trace database at the TraceBase website [21]. The *weather* application models the atmosphere around the globe using finite difference methods to solve a set of partial differential equations describing the state of the atmosphere.

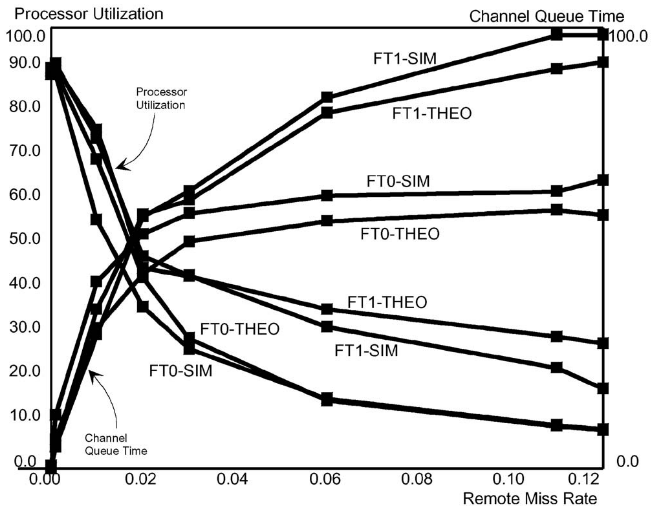

Fig. 7. System performance, Case N.

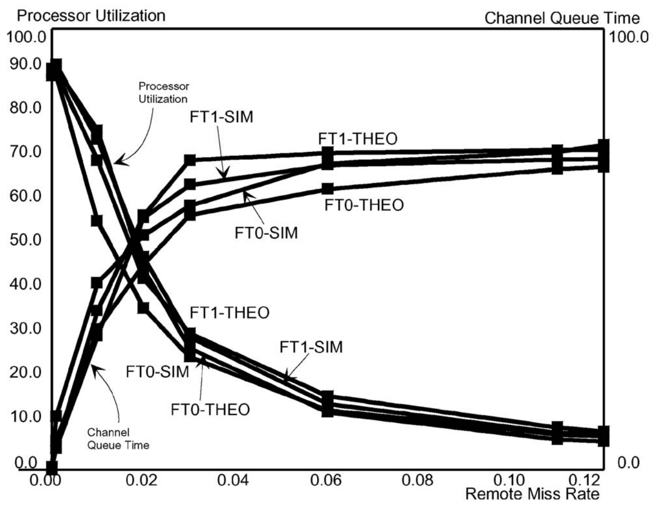

Fig. 8. System performance, Case S.

Speech uses a modified Viterbi search algorithm to find the best match between paths through a directed graph representing a dictionary and another through a directed graph representing spoken input in order to provide the lexical decoding stage for a spoken language understanding system. *Simple* is an application that models fluids using finite difference methods to solve a set of behavior-describing equations.

For the synthetic workloads used in our simulations, the memory access behavior of the application is specified by selecting the degree of locality and sharing of data between threads at different nodes. A memory reference generated by a thread can be to an address that falls within the host node's local memory, or to an address that falls within another node's memory, or to an address that is currently in the host node's cache. The memory reference may specify a block that is currently being accessed by or has recently been accessed by another node. Probabilities associated with these outcomes are used to control the degree of spatial locality for the memory references and the amount of interaction between threads: P_L is the probability that the reference is to the node's local memory, P_S is the probability that the reference is to any remote node's memory space and the referenced location has been accessed by some other thread in the past, P_N is the probability that the reference is to a neighboring node's memory, and P_C is probability that the reference is to one of the blocks already in the host node's cache. The neighbor of node X is node Y, such that Y is Home and X is Home2. P_N is the probability that a process in X may access a block that can be found in the local copy of Y's memory.

In the simulation results discussed in this paper, the behavior of an application is characterized by the degree to which a thread tends to access blocks in a remote node such that those blocks are either anywhere in that remote memory or are blocks that have been accessed in the past. Applications of interest can be described by the following cases: 1) $P_L = 0.01$, $P_S = 0.01$, and $P_N = [0.0, \dots, 0.10]$ indicating small degree of sharing between threads at different nodes and increasing degree of sharing between threads at a node and its neighbor (Case N), and 2) $P_L = 0.01$, $P_N = 0.01$, and $P_S = [0.0, \dots, 0.10]$ indicating small degree of sharing between threads at a node and its neighbor and increasing degree of sharing between threads at different nodes (Case S). Because P_L is relatively small, both cases indicate a high locality within each node. The performance of the architecture is evaluated in terms of processor and channel utilization, application execution time, checkpoint time, and mean channel queue waiting times. A system configuration is used with 32 nodes and 2 threads per node. Cache blocks have 64 bytes, data messages have 74 bytes, and messages with no data have 10 bytes.

7 Performance Comparison

Fig. 7 shows the system performance for case N, when no checkpoints are taken. The remote miss rate increases due to the increased sharing between a node and its neighbor as P_N increases. (The figure also shows the case where the remote miss rate approaches zero when both P_S and P_N become very small.) Since under the FT0 protocol, all activities related to fault-tolerance occur during the checkpoint, Fig. 7 shows the effect of protocol FT1 during regular application execution as compared to a system with no fault tolerance. Under the FT1 protocol, there is additional network activity between checkpoints mostly due to messages that maintain a consistent copy of the Home2 memory. Consequently, the messages tend to wait longer in the channel queue. However, as probability P_N increases, threads at each node tend to find the necessary data within the same node and become blocked less often, with a significant increase in processor utilization. Fig. 7 shows that applications that cause relatively higher sharing between pairs of nodes not only have fault tolerance for-free, but can benefit significantly by the additional replication of data in each (Home2) node.

When applications do not exhibit higher sharing between pairs of nodes, a smaller increase in processor utilization is observed, as shown in Fig. 8 for case S. The figure shows the remote cache miss rate increasing as each thread tends to access memory blocks in any other node (but with preference to blocks that have recently been accessed). A small increase in processor utilization is observed mostly due to the fact that threads may still find the necessary block in the (Home2) copy at the local node, although less frequently.

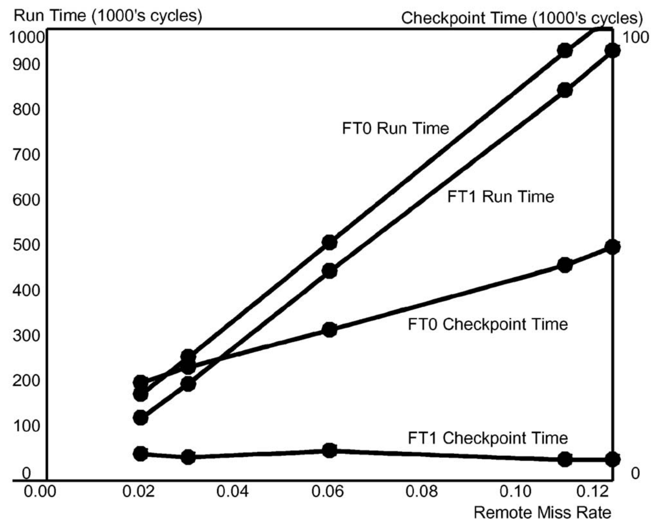

Fig. 9. Checkpoint effect on performance, Case S.

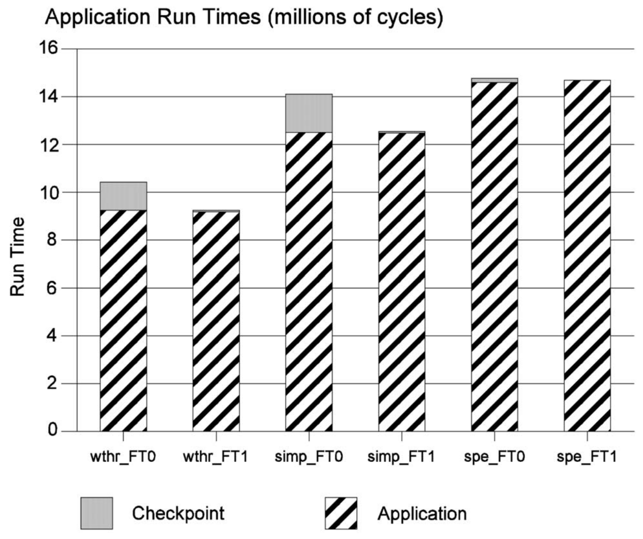

Fig. 11. Application traces: runtimes.

Even when an application does not exhibit higher sharing between pairs of nodes, the small increase in processor utilization results in a reduction of the application runtime as Fig. 9 shows for case S. Even more important is the reduction of time that is used to create checkpoints. Fig. 9 shows that as sharing between nodes increases (case S), more data blocks become modified and, therefore, under protocol FT0, more time is spent transferring the modified blocks to the backup nodes during the checkpoint creation. In contrast, under protocol FT1, the time spent to create a checkpoint is significantly reduced and remains unaffected by the increased sharing between nodes. A small fraction of the checkpoint time is spent while all nodes synchronize, and the remaining time is used by the caches writing back all modified blocks. The activity which only occurs when FT0 is used, is the additional message traffic between each node and its backup to update the recovery memory. It is accomplished under FT1 with no additional messages.

Fig. 10 shows similar runtime and checkpoint time measurements when an application does exhibit higher sharing between pairs of nodes (case N). The increase in processor utilization causes a significant reduction in the application runtime. The difference in application behavior has little effect on the checkpoint creation time under FT0, while under FT1 it is slightly smaller. Figs. 9 and 10 show that as the remote miss rate increases, the application runtime increases as expected. However, FT1 is preferable since it always results in smaller application run time than FT0. For some applications, the benefit is moderate (Fig. 9), while for other applications, the benefit is significant (Fig. 10). One of the reasons is that the checkpoint time in FT1 does not increase as the remote miss rate increases.

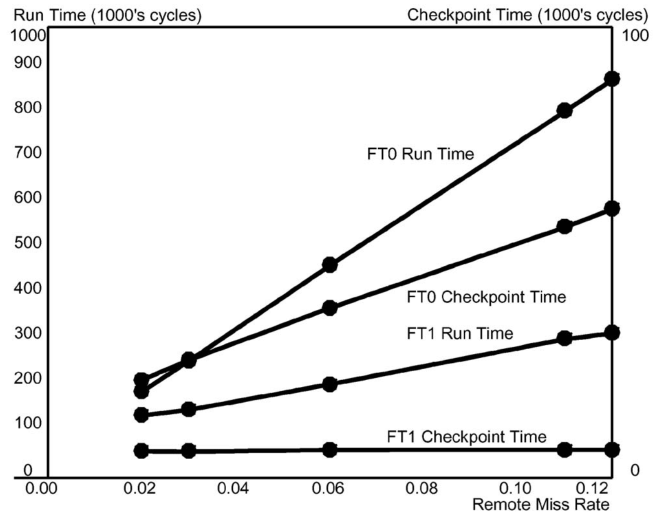

Fig. 10. Checkpoint effect on performance, Case N.

Fig. 11 provides a comparison of the protocols for the multiprocessor trace files. The vertical axis shows total runtime of each application using FT0 or FT1, in millions of clock cycles. Each bar shows the total run time: the upper part is the time spent on checkpoints, and the lower part is the real application runtime.

Since all FT-related activities in FT0 happen when a checkpoint is established, FT0 with no checkpoints corresponds to running the application with no fault tolerance. In Fig. 11, the lower parts of the bars relating to FT0 show that time. In FT1, there are FT-related activities happening between checkpoints. However, because the FT1 protocol can combine the FT and DSM activities, the real application runtime is actually less than the runtime obtained with a protocol with no fault-tolerance. For example, the lower part of wthr_FT1 is less than the lower part of wthr_FT0. The overhead, represented by the upper part of each bar, is quite reduced in FT1.

Applications perform better under FT1 rather than FT0 due to the elimination of the exchange of the Recovery Data Update messages during checkpoint creation, and the ability of a process to use the Home2 copy of local memory to fill a read request during normal runtime. The second mechanism provides an increasing benefit when data from a node's local memory is requested by its neighbor more frequently. The resulting benefit is the shorter time required to perform a local access of the data as compared to the time required to send a request message to the neighbor's directory controller and receive a data acknowledge message back.

The time required to create a checkpoint for both *weather* and *simple* is significantly lower for FT1 than FT0. In the *weather* application, the FT0 protocol results in a similar normal execution time as FT1, but the time to take a checkpoint causes the total execution time to be larger in FT0 than FT1. In the *simple* application, the FT1 protocol performs better because there are a large number of local accesses to different local data blocks and a large degree of sharing between the nodes, with requested data rarely found in the UNOWNED state, similar to case S described above. Both protocols perform similarly in the *speech* application due to the fact that a small number of data blocks are modified during the interval between checkpoints, resulting in a small checkpoint time in both cases. In addition, there is a very low probability of locality or sharing between the Home and Home2 nodes, which results in similar execution times during normal operation for both protocols.

In all three applications, the presence of hot spots has an adverse effect. In *speech*, hot spots are caused by frequent accesses to the data structures comprising two dictionaries. Instead of a single data block being continuously accessed, the entire dictionary structure is scanned repeatedly. Eliminating the hot spots would tend to increase the benefit of the FT1 protocol. This may be accomplished by dividing the dictionary data structure into smaller units that are distributed among the nodes. By arranging the dictionary data structures so that as the scan is performed, part of the data being requested by a node falls in the address space of its neighbor, the FT1 algorithm would reduce the number of request messages for that particular section of the dictionary scan.

8 Conclusion

Current advances in optical technology have made broadcast-based networks such as the SOME-Bus architecture realistic, highly-competitive candidates that promise to deliver very high performance. The power of the SOME-Bus is due to the fact that it requires no routing and no arbitration. No node is ever blocked from sending a message to any other node.

The SOME-Bus architecture contributes to the success of DSM by transferring messages with as little latency as possible and through efficient support of the coherence protocols. In addition, simple enhancements to the network interface and the cache and directory controllers allow the nodes to implement fault-tolerant protocols easily. The natural duplication of data blocks, necessary to support DSM and FT separately, is combined so that fault tolerance is accomplished without the penalty of increased network traffic. When application behavior allows it, the data blocks duplicated to support fault tolerance can be exploited by the DSM protocol resulting in even higher performance and reduced checkpointing time. Applications with a larger degree of sharing of data structures between all of the nodes in the system benefit substantially from the FT1 protocol, both in terms of normal execution time and in the time required to create checkpoints. In applications where there is an increased probability that data is shared between the Home and Home2 nodes, the FT1 protocol allows read misses on the Home2 node to be filled from the Home2 memory, which otherwise would have required a remote access. Simulation results show that even small increases in the probability of sharing data between Home and Home2 cause system performance to increase significantly.

Acknowledgments

The authors would like to acknowledge the substantial comments of the reviewers that greatly improved this paper. This material is based upon work supported by the US National Science Foundation under Grant No. EIA-9985971. Preliminary versions of this paper were presented at the IEEE Workshops on Fault-Tolerant Parallel and Distributed Systems, April 2002 and 2003.

References

[1] M. Banatre, A. Gefflaut, P. Joubert, and C. Morin, "An Architecture for Tolerating Processor Failures in Shared Memory Multiprocessors," *IEEE Trans. Computers,* vol. 45, no. 10, pp. 1101-1115, Oct. 1996.

[2] C. Calvin, "All-To-All Broadcast in Torus with Wormhole-Like Routing," *Proc. IEEE Symp. Parallel Distributed Processing,* pp. 130-137, 1995.

[3] R. Christodoulopoulou, R. Azimi, and A. Bilas, "Dynamic Data Replication: An Approach to Providing Fault-Tolerant Shared Memory Clusters," *Proc. Ninth Int'l Symp. High-Performance Computer Architecture,* pp. 203-214, Feb. 2003.

[4] L. Dong, B. Ortega, and L. Reekie, "Coupling Characteristics of Claddding Modes in Tilted Optical Fiber Gratings," *Applied Optics,* vol. 37, no. 22, pp. 5099-5105, Aug. 1998.

[5] B.D. Fleisch, "Reliable Distributed Shared Memory," *Proc. IEEE Workshop Experimental Distributed Systems,* pp. 102-105, 1990.

[6] B.D. Fleisch, H. Michel, S.K. Shah, and O.E. Theel, "Fault Tolerance and Configurability in DSM Coherence Protocols," *IEEE Concurrency,* vol. 8, no. 2, pp. 10-21, Apr.-June 2000.

[7] A. Grujic, M. Tomasevic, and V. Milutinovic, "A Simulation Study of Hardware-Oriented DSm Approaches," *IEEE Parallel & Distributed Technology,* vol. 4, no. 1, p. 74, 1996.

[8] D.L. Hecht, "Fault Tolerant Distributed-Shared Memory on a Broadcast-Based Architecture," PhD thesis, Drexel Univ., Dec. 2002.

[9] C. Ho, "Optimal Broadcast in All-Port Wormhole-Routed Hypercubes," *IEEE Trans. Parallel and Distributed Systems,* vol. 6, no. 2, pp. 203-205, Feb. 1995.

[10] Y.C. Hu, H. Lu, A.L. Cox, and W. Zwaenepoell, "OpenMP for Network of SMPs," *Proc. 13th Int'l Symp. Parallel and Distributed Processing,* pp. 302-310, 1999.

[11] C. Katsinis, "Performance Analysis of the Simultaneous Optical Multiprocessor Exchange Bus," *Parallel Computing J.,* vol. 27, no. 8, pp. 1079-1115, July 2001.

[12] A.M. Kermarrec, C. Morin, and M. Banatre, "Design, Implementation and Evaluation of ICARE: An Efficient Recoverable DSM," *Software Practice and Experience,* vol. 28, no. 9, pp. 981-1010, 1998.

[13] J.H. Kim and N.H. Vaidya, "Single Fault-Tolerant Distributed Shared Memory Using Competitive Update," *Microprocessors and Microsystems,* vol. 21, no. 3, pp. 183-196, Dec. 1997.

[14] Y. Li, T. Wang, and K. Fasanella, "Cost-Effective Side-Coupling Polymer Fiber Optics for Optical Interconnections," *J. Lightwave Technology,* vol. 16, no. 5, pp. 892-901, May 1998.

[15] C. Morin and I. Puaut, "Survey of Recoverable Distributed Shared Virtual Memory Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 9, pp. 959-969, Sept. 1997.

[16] D.V. Plant, M.B. Venditti, E. Laprise, J. Faucher, K. Razavi, M. Chateauneuf, A.G. Kirk, and J.S. Ahearn, "256 Channel Bidirectional Optical Interconnect Using VCSELs and Photodiodes on CMOS," *J. Lightwave Technolgy,* vol. 19, no. 8, pp. 1093-1103, Aug. 2001.

[17] S. Roy and V. Chaudhary, "Strings: A High-Performance Distributed Shared Memory for Symmetric Multiprocessor Clusters," *Proc. Seventh Int'l Symp. High Performance Distributed Computing,* pp. 90-97, 1998.

[18] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hung, L. Kontothanassis, S. Parthasarahy, and M. Scott, "Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote Write Network," *Proc. 16th ACM Symp. Operating Systems Principles,* pp. 170-183, 1997.

[19] O.E. Theel and B.D. Fleisch, "A Dynamic Coherence Protocol for Distributed Shared Memory Enforcing High Data Availability at Low Costs," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 9, pp. 915-930, Sept. 1996.

[20] J.G. Turk and B.D. Fleisch, "DBRpc: A Highly Adaptable Protocol for Reliable DSM Systems," *Proc. 19th IEEE Int'l Conf. Distributed Computing Systems,* pp. 340-348, 1999.

[21] http://tracebase.nmsu.edu/tracebase.html, 2004.

Constantine Katsinis received the BS degree from the Polytechnic University of Athens, Greece, and the MS and PhD degrees from the University of Rhode Island, Kingston, Rhode Island, in electrical engineering. He was an associate professor at the University of Alabama in Huntsville until 1998. Since then, he has been an associate professor at Drexel University. His research interests include parallel computer architectures, microprocessor systems, image processing, and performance modeling. He is a member of the IEEE and the IEEE Computer Society.

Diana Hecht received the MSE and BSE degrees from the University of Alabama in Huntsville and the PhD degree in computer engineering from Drexel University in 2002. After graduating from Drexel, she went to work for Rydal Research and Development, Inc. Her research interests include parallel and distributed computing and systems, fault-tolerant systems, computer networks, and switch architectures. She is a member of the IEEE and the IEEE Computer Society.