Distributed Virtual Memory Consistency Protocols: Design and Performance

Aamod Sane Ken MacGregor Roy Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Avenue, Urbana, Illinois 61801–2987

Abstract

An object-oriented framework for the design of distributed virtual memory consistency protocols is presented. It is shown that custom designed protocols for different applications are easy to construct and use with this framework. Consistency protocols are shown to be useful in implementing atomic update, and in controlling assignment of pages to processes. Finally, experimental results are presented.

1 Introduction

Distributed virtual memory (*DVM*) exploits traditional virtual memory mechanisms to support a shared memory style of multiprocessor programming on a network of computers. In principle, parallel programs written for shared memory machines may be ported to a loosely coupled environment without change.

To maximize the number of parallel activities possible, local copies of the shared data are maintained on the machines involved. Some form of consistency among the various copies is guaranteed, for example, that of shared memory in a shared memory multiprocessor. Consistency protocols control the replication and invalidation of data within the system to support the required definition of consistency. Since they control page movement, they can be used to control the page assignment to network nodes.

In this paper, we study the *Choices* framework for the design of consistency protocols. We also show that these protocols can be gainfully employed to control page assignment. Finally, experimental results for some sample protocols are presented.

*This work was supported in part by NASA under grant number NSG1471 and by AT&T METRONET.

2 Overview of Choices

Choices [1, 8] began as an investigation of the use of class hierarchies and object-oriented design for the construction of multiprocessor operating systems. All operating system concepts and components are implemented within the framework of a class hierarchy. Subclasses are used to encapsulate machine dependencies and to separate mechanisms from policy decisions. *Choices* is implemented in over 70,000 lines of C++ and includes code for virtual memory management, interrupt and exception handling, parallel processing, file systems, I/O, and networking. C++ supports class hierarchies and object-oriented design without sacrificing efficiency.

Choices is being used in the Tapestry[1] laboratory [3] to support the investigation of parallel applications and distributed systems in a heterogeneous environment. It currently runs on the Encore Multimax shared-memory multiprocessor and is being ported to the Macintosh II, HP Precision, and Intel i386 machines.

2.1 Virtual Memory in Choices

A *Choices* process [7] executes in a Domain [2], which is a mapping of a virtual address space to MemoryObjects. A MemoryObject is a logical collection of data. It is made accessible to the process by its MemoryObjectCache. A MemoryObjectCache maintains its own machine independent physical memory management

[1]Tapestry is funded by an NSF CISE grant.

information. A MemoryObject can be mapped into multiple Domains, providing shared memory.

One or more processes can execute within one Domain. *Choices* processes are lightweight, and within a Domain, context switching is inexpensive.

After a page fault, the MemoryObject corresponding to the faulting virtual address is determined using the Domain. The RepairFault method is invoked on the MemoryObject to get the data at the virtual address. In turn, RepairFault is invoked on the MemoryObjectCache. A local caching strategy, encapsulated entirely within the MemoryObjectCache, is used to repair the fault. Subclasses of the MemoryObjectCache embody the ways in which a fault is repaired. This includes traditional paging, simple alteration of access (e.g. to implement copy on write), and getting the page from across the network (for *DVM*).

3 Distributed Virtual Memory

The *DVM* implementation[5] consists of two important parts: extensions to the virtual memory class hierarchy, and the consistency protocol implementation[2].

The important classes implementing *DVM*,DistributedMemoryObjectCache and PageRecord are considered below.

3.1 Class DistributedMemoryObjectCache

An instance of DistributedMemoryObjectCache provides a local physical memory cache for the copy of the shared data on a networked node. These DistributedMemoryObjectCaches form a peer group. Each DistributedMemoryObjectCache is responsible for locating and retrieving pages from its peers in order to repair virtual memory faults generated by the Process(es) on its node. Local copies of data can be paged on their respective nodes. This activity is also managed by the DistributedMemoryObjectCache.

3.2 Class PageRecord

The PageRecord class is the hub of the *DVM* implementation. Besides the traditional VM information inherited from the VM Page Record, page state and other information required for maintaining coherence is kept in instances of this class. The *Choices* consistency protocols are defined using state machines that are implemented in the PageRecord. The methods of PageRecord and its subclasses correspond to the generation of and responses to events such as

- messages between nodes,
- timeouts,
- local read/write accesses or
- process termination.

Existing protocols can be combined (using multiple inheritance) or subclassed to create new protocols.

4 The Consistency protocols

4.1 The Basic Choices Protocol

The basic *Choices* protocol[5, 6] is defined in the class DVMCompletePageRecord. It is designed to avoid the overhead associated with a heavyweight network protocol, therefore, it assumes a low-level, unreliable[3] datagram service, handles page to packet assembly/disassembly as well as recovery from lost packets.

Consistency is maintained using a *single-writer / multiple-readers* discipline.

When writable, a page resides on one machine. Read requests from other machines make the page read only, and copies of the page are sent to the requesters. A list of copy holders is maintained at the machine that originally had write access. This machine is designated as the *owner* of that page. A subsequent write request will be serviced by the owner, invalidating the readable copies and giving the writable page to the requester. The requester then becomes the new owner of that page.

Reader and Writer processes can dynamically join or leave the group of processes using *DVM* .

This protocol does not specifically provide any support to

1. guarantee page assignment, i.e.
   - retain pages till sufficient work may be done with them.
   - order page usage among processes to enforce data dependencies.
2. guarantee atomic update.

Without support within the protocol, the above can be achieved at the additional cost of unnecessary application knowledge, inefficiency, and possibly fatal susceptibility to untrustworthy processes. The *Delay* and *Lock* protocols (Sec 4.3, 4.4) provide suitable support.

[2] Other parts include the *DVM* setup machinery, basic network communications etc.

[3] By unreliable, we mean that delivery is not *guaranteed*, though many local area networks are reliable in practice.

4.2 Characteristics Determining Performance

The time overhead resulting from the protocol is a function of control and data packets generated and data structure processing. This may be represented as:

$$O = N.p_k + M.p_g + F + V.m \quad (1)$$

Where:

O: protocol overhead

p_k: Total number of packets

p_g: Packets containing data

N: Network transmission cost per packet

M: Memory usage cost for data transfer per packet

F: Fixed data structure processing costs that do not depend directly on protocol events

V: Data strucuture processing cost related to maintaining information about copy holders.

m: number of machines (maximum number of copies)

Not all of this overhead reflects directly on the total time taken by the application, since some events are handled concurrently with the application.

Performance also depends on page assignment. The basic protocol determines the overhead of events.

For the basic events Read, Write, UpgradeO, UpgradeNO (change from read copy to write when page owner and not page owner, respectively) maximum overhead is:

Read Read request (1 packet) + Read data transfer (3 packets)[4] = $4N + 3M + V$

Write Write request (1 packet) + Copy invalidations (2(m-2)[5] packets) + Domain flush (Fixed cost) + Write data transfer (3 packets) + Ack (1 packet) = $5N + 3M + F + (m-2)(V + 2N)$

UpgradeO Copy invalidations (2(m-1) packets) = $(m-1)(V + 2N)$

UpgradeNO Upgrade request (1 packet) + Copy invalidations (2(m-2) packets) + Domain flush (Fixed cost) + Upgrade notification (1 packet) + Ack (1 packet) = $3N + F + (m-2)(V + 2N)$

[4] On Multimaxes, page size is 4k, packet size 1.5k

[5] 0 if m $\leq$ 2

4.3 A Locking protocol

This protocol guarantees atomic update by denying requests for a locked page. If the basic protocol and test-and-set locks are used, it is not possible to *guarantee* that a page will remain; circumvention is possible.

A response message **Retry** is returned to any requesters. Processes receiving this message can either sleep or send the request again. This defines the lock as sleep or spin lock respectively. For a sleep lock, a queue of requesters is kept to send the wakeup.

Sleep locks minimize network loading but centralizes wakeup information and block waiting processes. A spin lock provides non-blocking synchronization. Response to **Retry** can be handled differently on each requester. Network loading is minimal if the retry interval is sufficiently large.

This protocol can also be used as an ordinary lock to guarantee order of page use to satisfy data dependencies. Such usage, however is restricted to processes executing in different Domains.

It is also possible to lock pages for a given amount of time, to satisfy the page requirements of an application (when they can be predicted beforehand).

4.4 A Delay protocol

The *Delay* protocol is a method for retaining pages for a sufficient time without application knowledge.

In the basic protocol, a request for a writable copy of the page is serviced immediately. Two (or more) processes writing simultaneously could cause a page to thrash from machine to machine, without any useful work being done.

A *threshold time*, after which external requests for copies of the page are serviced, can improve the ratio of useful time to page transport time in all except pathological cases. Requests arriving during this time are queued and serviced later.

The threshold time value is decided by the current activity within the page and the number of requests for that page in order to generate a page assignment that reduces page movement and increases page usage.

Threshold time also reduces possibility of live-lock, where a page is shuttled between some machines, and others never get access.

We are in the process of studying applications to evaluate the impact of this protocol and to determine a suitable heuristic to decide the threshold time value.

4.5 Variations

The protocols can be altered to trade off characteristics such as resiliency to packet loss, network loading, etc.

For example, in the basic protocol, only an approximate knowledge of the owner is kept on all nodes except those that have read/write access. Thus, requests from a node that does not have a copy of the page can end up at a node that is not the actual owner. One of two things can be done now: forward the request[4, 6] to the current value of owner (which is approximate) until the actual owner is reached, or indicate the possible owner to the requester for an explicit send by the requester.

In the former strategy, the number of messages is less than in the latter. However, the latter is more resilient to network packet loss.

5 Empirical Results

Matrix multiplication and Producer Consumer were selected as applications on the two extremes of page conflicts. Matrix multiplication partitions very well, and Producer Consumer exhibits page conflicts.

The applications were run on 2 Encore Multimaxes each with 4 processors connected by Ethernet[6].

5.1 Matrix Multiplication

A simple 3 loop matrix multiplication program was used. In the case of the lock protocol, each page was locked when calculating the row on that page. Figure 1 illustrates timings for 256x256 matrix multiply. Table 1 shows the timings for the 2 protocols. The results are summarized below:

- Matrix multiplication time for *DVM* is comparable to that of shared memory
- *DVM* offers access to a greater number of processors, and gives better performance when number of processes on a single machine exceeds number of processors
- Paging behaviour is altered drastically by access patterns and shows up in the "kij" versus "ijk" values.
- The locking algorithm gives slightly better performance by saving some page transfers.

5.2 Producer Consumer

The producer locks a buffer and fills it, and releases the lock. The consumer then acquires the lock and empties the buffer. The processing time is enough to force **Retry**.

Table 2 illustrates the timings for Producer Consumer.

The *Lock* protocol is significantly more efficient than the basic protocol even when spin locks are used — 33.4% of the basic protocol time for a buffer up to 4k in size and 55.4% of the basic protocol time for a buffer of 10k. When the buffer size increases to over 60k the basic protocol is more efficient. The amount of network traffic generated by the *Lock* protocol is also less. For larger packet sizes and longer fill/empty processing times, traffic could be further reduced by using a sleep lock.

6 Analysis of Performance

For *DVM* to achieve the efficiency required for parallel applications, the transfer time for a page over the network should be close to disk swap time, which on the Encore Multimax in *Choices* is 20ms. In analyzing the relative contributions of processor speed, memory speed and network transmission to the page transfer rate, it was found that the memory access time followed by the network setup time were by far the largest components. Some of this resulted from the Encore Ethernet driver performing multiple memory copies per packet transferred, while another contributing factor was the page assembly/disassembly resulting from the 1.5k Ethernet packet restriction.

Table 3 shows present values and the effect of 50% less memory overhead, network overhead, and 50% faster processor[7]. Only in the last case is the desired performance achieved. This would require a communication transfer mechanism such as a hyperchannel with a packet size at least that of a page and DMA from memory to the communication controller. Increases in processor speed, and network transmission speed would only provide marginal improvements.

7 Conclusions

Performance of applications based on *DVM* is sensitive to their locality properties. To get the best performance, it is necessary to choose an appropriate consistency protocol. Our paper demonstrates a versatile framework for the design of protocols. It is easy to specialize existing protocols or combine them to get a custom designed protocol. The rest of the VM system supports the creation of independent policy decisions for different shared areas.

[6] Choices was designed on the Multimaxes since they are multiprocessors. Workstation ports are in progress and a testbed of more machines should be available soon.

[7] Our present processors are 16Mhz NS32332s

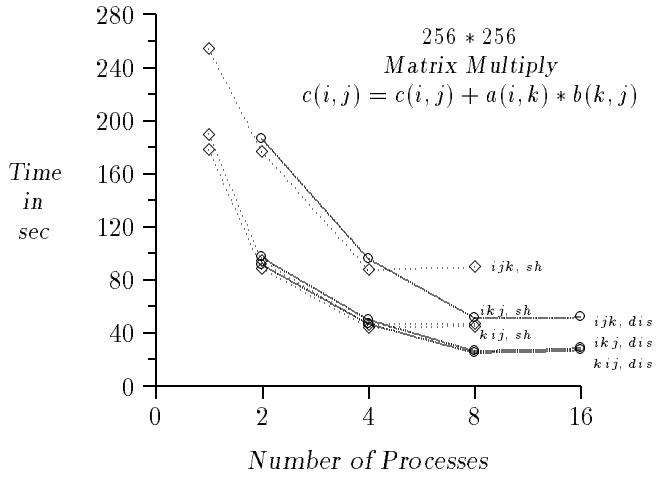

Figure 1: Speedup and Comparative Performance with Shared and Distributed Memory

Table 1: Comparative timings (in sec, ±1) for matrix multiplication

No. of Processes	Shared(kij)	Distributed(kij)		Shared(ijk)	Distributed(ijk)	
		Basic	Locking		Basic	Locking
2	89.2	91.9	90.98	176.93	186.71	181.01
4	44.56	47.45	47.36	88.12	96.47	92.23
8	45.50	25.66	24.96	90.18	51.92	51.03

Table 2: Comparative timings (in ms, ±2) for Producer Consumer

Buffer Size(pages)	Basic (test-and-set)		Locking	
	Time(ms)	No. of Packets	Time(ms)	No. of packets
1	219.8	28	83	10
3	325.1	40	181.7	30
10	690.3	82	551.6	100

Table 3: Basic Operation timings (in ms, ±1) for varying system component speeds(for 2 machines)

Event	Measured	Memory	Network	Processor	All
Read	39.9	27.8	30.4	33.7	18.2
Write	48.2	36.2	36.4	41.2	41.2
UpgradeO	19.4	19.4	12.3	18.4	11.2
UpgradeNO	11.1	11.1	6.3	11.0	6.2

We also show that consistency protocols can be used to achieve more than consistency.

The *Lock* protocol provides a way of guaranteeing atomic update, by preventing movement of the page. These page locks can also be used as distributed locks, however, the use is limited to between Domains. When used as distributed locks, they are cheaper than test-and-set locks. This protocol can also be used to "assign" pages, when predictable by the application.

The *Delay* protocol attempts to optimally assign pages based on page usage, retaining pages on a machine. This is achieved without any knowledge of the application. The protocol also reduces the possibility of livelock problems.

Other protocol variations to trade off the characteristics that affect performance are also indicated.

The experimental results reported are limited due to the current non-availability of *Choices* ports to workstations. Ports to HP Precision machines, MacII, and ATT6386 machines are nearing completion. We then intend to investigate the behavior of distributed applications using the protocols mentioned.

References

[1] Roy Campbell, Gary Johnston, and Vincent Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[2] Roy Campbell, Vincent Russo, and Gary Johnston. A class hierarchical, object-oriented approach to virtual memory management in multiprocessor operating systems. Technical Report UIUCDCS–R–88–1459, Department of Computer Science, University of Illinois at Urbana-Champaign, 1988.

[3] Roy H. Campbell and Daniel A. Reed. TAPESTRY: Unifying shared and distributed memory parallel systems. Technical Report UIUCDCS–R–88–1449, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1988. Also Tapestry Technical Report No. TTR88–1.

[4] A. Forin, J. Barrera, and R. Sanzi. Machine–independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

[5] Gary M. Johnston and Roy H. Campbell. An object oriented implementation of distributed virtual memory. In *Proceedings of the USENIX Workshop on Distributed and Multiprocessor Systems*, pages 39–58, September 1989.

[6] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 229–239, 1986.

[7] Vincent Russo, Gary Johnston, and Roy Campbell. Process management and exception handling in multiprocessor operating systems using object-oriented design techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1988. Also Technical Report No. UIUCDCS–R–88–1415, Department of Computer Science, University of Illinois at Urbana-Champaign.

[8] Vincent F. Russo. *The Design of an Object-Oriented Operating System.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1990 (forthcoming).