PROJECT ADMINISTRATION DATA

OCA contact: Steven E. Watt  894-4820

Sponsor technical contact

BENJAMIN W. WAH
(202) 357-7853
NATIONAL SCIENCE FOUNDATION
CISE/MLP
WASHINGTON, D.C. 20550

Defense priority rating: N/A
Equipment title vests with: Sponsor
SUN 3/60 H-4P10

Sponsor issuing office

SHARON GRAHAM
(202) 357-9621
NATIONAL SCIENCE FOUNDATION
DGC/CISE
WASHINGTON, D.C. 20550

DNR resident rep. is ACO (Y/N): N/A
N/A supplemental sheet

Administrative comments -
INITIATION. RESEARCH INITIATION AWARD. INDIRECT COSTS RESTRICTED TO 10% AS A CONDITION OF THE GRANT.
GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

Closeout Notice Date 05/23/91

Project No. C-36-659
Project Director RAMACHANDRAN U
Center No. R6513-0A0
School/Lab COMPUTING
Sponsor NATL SCIENCE FOUNDATION/GENERAL

Contract/Grant No. MIP-8809268
Prime Contract No.

Title DISTRIBUTED OPERATING SYST & MACHINE ARCHITECTURE:REDUCING THE SEMANTIC G
Effective Completion Date 901130 (Performance) 910228 (Reports)

Closeout Actions Required:  

<table>
<thead>
<tr>
<th>Item</th>
<th>Y/N</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final Invoice or Copy of Final Invoice</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Final Report of Inventions and/or Subcontracts</td>
<td>Y</td>
<td>910523</td>
</tr>
<tr>
<td>Government Property Inventory &amp; Related Certificate</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Classified Material Certificate</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Release and Assignment</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

Comments: BILLING VIA NSF LINE-OF-CREDIT. 98A SUBMITTED WITH FINAL REPORT SATISFIES REQUIREMENT FOR PATENT REPORTING.

Subproject Under Main Project No. _______________________
Continues Project No. _______________________

Distribution Required:

<table>
<thead>
<tr>
<th>Distribution Required</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Director</td>
<td>Y</td>
</tr>
<tr>
<td>Administrative Network Representative</td>
<td>Y</td>
</tr>
<tr>
<td>GTRI Accounting/Grants and Contracts</td>
<td>Y</td>
</tr>
<tr>
<td>Procurement/Supply Services</td>
<td>Y</td>
</tr>
<tr>
<td>Research Property Management</td>
<td>Y</td>
</tr>
<tr>
<td>Research Security Services</td>
<td>N</td>
</tr>
<tr>
<td>Reports Coordinator (OCA)</td>
<td>Y</td>
</tr>
<tr>
<td>GTRC</td>
<td>Y</td>
</tr>
<tr>
<td>Project File</td>
<td>Y</td>
</tr>
<tr>
<td>Other</td>
<td>N</td>
</tr>
</tbody>
</table>
DISCLAIMER:

This document has been proofed and its original formatting has been retained.
September 13, 1989

Dr. Benjamin Wah
Program Director
Microelectronic Systems Architecture
Division of Microelectronic Information Processing Systems
National Science Foundation
Washington, DC 20550

Re: NSF Award No. MIP-8809268

Dear Dr. Wah:

This letter is the annual progress report for the referred grant.

As proposed, during the first year of the grant I have performed research on investigating architectural support for object-based and message-based operating systems. This research has resulted in identifying the features of a memory management unit for object-based systems (papers 1, 2, and 3), issues in managing distributed shared memory (papers 4, 5, and 6), and preliminary investigation of primitives for synchronization in shared memory multiprocessors (paper 7). Research work done under this grant has produced the following publications:


I have also presented the results from this research at the International conference on parallel processing, University of Wisconsin - Madison, Digital Equipment Corporation, Boston, and University of Southern California, Los Angeles. I will be making presentations later this month and next at the International conference on computer design, and International computer software and applications conference.

In the second year of the grant I intend to continue investigation into the synchronization primitives for shared memory multiprocessors, quantitative evaluation of memory management units for object-based systems, and investigation of message-passing versus shared memory alter-
natives for structuring distributed systems.

Best regards,

Umakishore Ramachandran
Assistant Professor
Phone: (404) 894-5136

encl:
Programming with Distributed Shared Memory*

Umakishore Ramachandran
M. Yousef Amin Khalidi

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Ga 30332-0280 USA

rama@gatech.edu
yak@gatech.edu

Abstract

In a distributed system, remote services may be provided by either remote procedure call (RPC) mechanism, or by paging in the required memory segments and performing the services locally. The latter approach, termed Distributed Shared Memory (DSM) has several benefits given the current trend of structuring computing systems using diskless computational servers (workstations) and data servers (file servers). We propose a set of distributed shared memory mechanisms that handle network-wide memory management for an object-based system. We discuss the implementation of these mechanisms and provide examples of their use in implementing the programming language Linda, process migration, two-phase commit, and a distributed game.

*This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.
1 Introduction

The resources in a loosely-coupled distributed system can be viewed as a collection of objects. The name space of all objects constitute a "global distributed shared memory". In this paper, we suggest mechanisms for supporting the abstraction of a globally distributed shared memory, and applications using these mechanisms. These mechanisms were designed in the context of the Clouds [BHK+88, DLA88] object-based distributed operating system.

The paper is organized as follows: We start with an overview of Ra—a kernel for Clouds and its relation to the shared memory module which we call distributed shared memory controller (DSMC). In §3 we discuss related work, and in §4 we discuss the status of our implementation. The three sections that follow present applications that use the DSMC mechanisms in implementing the Linda parallel programming language [Gel85], process migration, and 2-phase commit. Section 8 investigates the use of weak memory coherence semantics in the implementation of a distributed game. Section 9 evaluates the use of distributed shared memory vs. remote procedure call for invoking remote objects. Finally, conclusions and future work follow in §10.

2 Ra and DSMC

Ra [BHK87, BHK+88, BHK+89] is an operating system kernel designed to be the nucleus of Clouds operating system. It currently runs on the Sun-3 architecture. Ra defines and manages three primitive abstractions: segment, virtual space, and isiba. Segments serve as containers of data and may be viewed as uninterpreted sequences of bytes. The contents of a segment may only be accessed when the segment is attached to a range of virtual addresses. Segments persist until explicitly destroyed. Each segment resides in a partition that is responsible for providing backing store for the segment. A partition is an entity that realizes, maintains, and manipulate segments (see §4).
Virtual spaces abstract the notion of an addressing domain. A Ra virtual space is described by a segment called the virtual space descriptor that contains a collection of windows. Each window is a data structure that identifies the segment that backs a range of addresses in the virtual space. A virtual space is composed (or built) by a sequence of attach operations, each of which defines one of the virtual space's windows. An object is a virtual space (called O space) that consists of code and data segments. The code segment of an O space has entry points that can be invoked by user processes.

Ra isibas are an abstraction of the fundamental notion of computation or activity and can be thought of as lightweight processes. Isibas may be used as daemons within the kernel or they may be associated with a virtual space (a P space) to implement a user process.

The machine virtual address space consists of three distinct regions that are called K, O, and P hardware spaces for kernel, object, and process, respectively. The kernel is always mapped in the K hardware space. (System objects, which we discuss in §6, are also mapped into the K space, but may be installed and removed dynamically.) A Process is mapped into the P hardware space and unmapped on context switch. A process consists of an isiba and a virtual space (P space). The object in which a process is currently executing is mapped into the hardware O space.

Local object invocation involves mapping the required memory segments of the object into the address space of the invoking process by installing the object as the current O space with the process's P space. The current trend in structuring distributed systems is to use a collection of diskless computational servers (workstations) and a few data servers (file servers). In such an environment, the code and data for the (local) invocation has to be paged-in from the data server. Further, for remote object invocation we have one of two choices: The first choice is to perform the computation at the node where the object resides (remote procedure call). The second choice is to make the invocation appear local by bringing in the segments required for the invocation. While we have to support the former
for immovable objects (such as an object that reads disk blocks), we believe that the latter may be a better choice for movable objects. There are two reasons to support this belief:

- the principle of locality [Den72] that suggests an invocation (or other invocations in the same object) may be repeated
- the reduction in computational overhead due to the elimination of slave process management to support remote invocation at the node where the object resides [Kha89].

Each host has a distributed shared memory controller (DSMC) that together with the network interface assists the host in mapping remote memory segments into virtual address spaces on the local host. The system memory is (logically) partitioned into two parts: One part (object memory) is for housing the segments of locally created objects; the other part (network cache) is for caching segments from remote objects. There is a concept of ownership whereby the node where a segment is created (the owner node) is responsible for guaranteeing the consistency of the segment.

The basic operations provided by the DSMC are get, and discard. The get operation is used to fetch a segment from its owner and to map it locally, while discard is used to return a segment to its owner. The DSMC provides synchronization primitives as separate operations (P and V semaphore operations), or as combined access and lock operations using the get and discard primitives.

Using the get primitive a segment may be acquired in one of four modes: read-only, read-write, weak-read, or none. Read-only mode signifies non-exclusive access but guarantees that the segment will not change until the node explicitly discards the segment. Read-write mode signifies exclusive access with a guarantee that the segment will not be thrown away until the writer explicitly discards the segment. Weak-read mode signifies non-exclusive access with no guarantee whether the segment will change or not. None mode signifies exclusive access with no guarantee whether the segment will be thrown away or
A Ra partition called DSM partition uses the DSMC primitives to communicate with remote DSM partitions and is responsible for acquiring the current copy of a segment when such a segment is accessed. (The DSM partition is also responsible for communicating with the local disk partition, if any.) Using distributed shared memory, any node can access any segment in the system, regardless of its location. Before a segment can be accessed, it has to be activated. Similarly, virtual spaces are activated before use (i.e. the segment that describes the virtual space is activated).

The DSMC and Ra provide mechanisms that enable the system object programmer to implement various operating systems functions. DSMC and Ra embody very little policy decisions; it is up to the system programmer to provide policies that control and recover the system resources. In this paper, the operating system refers to the set of (system and user) objects that use the mechanisms provided by Ra and the DSMC to implement resource managers, name-servers, process-control managers, etc.

3 Related Work

Our work is built on the large body of work that exists in maintaining cache coherence in multiprocessors [AB86]. Li [LH86] proposes a shared virtual memory system for a network of machines connected through a token-ring network, while Cheriton [Che86a] advocates problem-oriented memory as the basic concept for structuring distributed systems. The novel feature of our approach is the exploitation of process synchronization to simplify consistency maintenance. We unify synchronization and data transfer in a single primitive thereby reducing the total number of messages exchanged in the system. Further, these primitives are integrated into an object-based operating system kernel (Ra), thereby assuring an efficient implementation.

We have investigated the use of our mechanisms for remote object invocations [RAK88,
Kha89], and have discussed the design and implementation of a native kernel that integrates distributed shared memory for network-wide memory management [BHK87, BHK+88, BHK+89]. Distributed shared memory in our environment plays a role similar to network file systems in conventional systems (e.g. NFS [SGK+85], RFS [BLMY87], and the Sprite file system [NWO88]).

4 Implementation of DSMC

We have implemented the DSMC as a software module that consists of approximately 3000 lines of C++ [Str86]. Figure 1 shows the organization of the DSMC implementation on Ra. The boxes in the figure denote system objects. System objects encapsulate necessary and/or useful operating system services and resource managers that have direct access to the Ra kernel, but are nonetheless outside the kernel. System objects are trusted software modules that are loaded dynamically in the system space (K space). Examples of system objects include device drivers, resource managers, user-level object support, and partitions. The DSMC cooperates with remote DSMC's to implement the distributed shared memory primitives. (See [RAK88] for detailed description of the DSMC algorithms.) DSM Partition is a Ra partition that provides the kernel with the ability to create/destroy and activate/deactivate segments, page-in/page-out portions of segments, and semaphore P/V operations. The DSM partition decides if a segment is owned by the local node or a remote node. It uses the Disk Partition to access local segments, and cooperates with the DSMC to access remote segments. Transaction abstraction layer (TAL) is a simple transaction-oriented communication protocol that provides reliable multi-packet request/response messages. It is used by the DSMC to communicate with remote DSMC's. TAL protocol is similar to other transaction-oriented protocols such as VMTP [Che86b]. However, it is much simpler than VMTP since it is tailored to our application domain. The DSMC algorithms require simple request/response messages only (possibly with message forwarding). Therefore, it is possible to substitute other transaction-oriented protocols for TAL. The
Disk Partition maintains segments owned by the local node on the local stable storage (if any).

The DSMC implementation executes on top of Unix (as a user process) or Ra. The operating system dependencies are isolated in a few C++ classes. The organization of the DSMC implementation on Unix is shown in Figure 2. On Unix, TAL runs as a user process that uses SUN's Network Interface Tap (NIT) [Sun86] to receive packets from the net and to route them among a set of clients and servers. The DSMC code is linked-in with client code. DSMC code is also linked-in with server code that uses the Unix file system to store segments. Implementation of DSMC on Unix and Ra serves three purposes:

1. The Unix environment makes it easy to test and verify the DSMC and TAL protocols.

2. The Unix file system is available for use as permanent store for segments. Ra executes on diskless Sun-3 workstations with backing store provided by Unix machines.

3. The strength of Unix is the rich program development environment that it provides. The strength of Clouds is transparent management of distributed data and computation. Providing interoperability between Unix and Clouds is one of our design goals. DSMC implementation on Unix and Ra serves this purpose. System and user objects are developed on Unix and demand-paged to Ra via DSMC mechanisms.

The Unix implementation of the DSMC and TAL is complete, and we are in the process of tuning it. Preliminary measurements show that a get() request for an 8K segment (which is the physical page size for Sun-3 architecture) costs around 43 ms between two Sun-3/60 using a 10 Mbits ethernet (when the segment is in memory at the owner node). The Ra implementation is awaiting the completion of an ethernet driver for Ra. We expect that a Ra kernel implementation of a get() request for an 8K segment will not exceed 10 ms.
5 Implementing Linda

Linda [Ge185, CG86] is a parallel programming language in which processes communicate through a logically shared structured associative memory called tuple space (TS) [Ge185]. There are three main operations that a process can perform on TS:

1. The primitive `out(t)` adds tuple `t` to TS.

2. The primitive `in(s)` causes some tuple `t` that matches template `s` to be removed from TS. If a matching tuple is not available at this time, then the invoking process is blocked until such a tuple is added to TS. If more than one tuple matches the template, one is chosen arbitrarily.

3. The primitive `read(s)`, is similar to `in(s)`, with the difference that the matching tuple is not removed from TS.

The Linda machine [ACGK88] consists of several processors arranged as a grid. Programs running on each processor issue `out(t)`, `in(s)`, and `read(s)` requests. `out(t)` requests broadcast the tuple `t` across a row of processors, while `in(s)` and `read(s)` broadcast search requests across a column. Each node has a local memory that maintains a subset of tuples in the TS. On each node, tuples are allocated in units of 16 bytes and a sufficient number of these units are attached to a tuple hanger to store all the data in the tuple. Tuples are chained together in *tuple groups* as identified by the Linda compiler [Car87]. The compiler extracts grouping information at compile time that enable processors to look only in specified groups when looking for a match to `in(s)` or `read(s)` broadcast requests. This grouping information is kept in an association table.

Due to the requirement of a global strongly consistent shared memory, an implementation of Linda in a distributed system has been non-obvious. In the rest of this section, we discuss an implementation of Linda using the DSMC primitives.
We allocate tuple groups in different segments: one group per segment \((S_1, \ldots, S_n)\) where \(n\) is the number of groups. Within each segment, tuples are stored in a format similar to the one used by the Linda compiler [Car87]. A read-only segment \((S_a)\) holds the association table generated by the compiler. An additional segment \((S_b)\) is used to hold information about blocked processes waiting for a match to \(\text{in}(s)\) and \(\text{read}(s)\) requests. A semaphore segment \((S_e)\) provides semaphores for queueing blocked processes. Each entry in \(S_b\) has the following fields:

- **template** /* to be filled in */
- **semaphore** /* address of a semaphore from \(S_s\) */
- **request_type** /* in or read */
- **match** /* true or false */

In addition, \(S_b\) contains a simple bit-map \((S_b\)-bitmap) of used/free semaphores. The \(\text{in}()\) and \(\text{out}()\) operations acquire semaphores by finding an unused semaphore and marking it as "in use", and return semaphores by marking them as "unused". The semaphores themselves reside in the semaphore segment \(S_e\), and the \(P()\) and \(V()\) operations on them are provided by the DSMC. Note that whereas \(\text{out}()\) and \(\text{in}()\) requests acquire the tuple segments in exclusive mode, \(\text{read}()\) requests acquire them in non-exclusive mode allowing other readers to look for a match concurrently. The primitives are implemented as follows:

- **out\((t)\)**

  \(S_b\) is first searched to see if there are any blocked \(\text{in}()\) or \(\text{read}()\) requests that match the tuple \(t\) being inserted in \(TS\). If there is at least one such \(\text{in}()\) request, the contents of \(t\) are stored in \(S_b\) and the waiting process is awakened. Otherwise, tuple \(t\) is inserted in one of the tuple groups (as determined by the association table \(S_a\)). If there are any \(\text{read}()\) requests that match \(t\), each waiting process is awakened. The algorithm for \(\text{out}()\) is as follows:

\[
\text{Get}(S_b, \text{read-write})
\]
\[
\text{success} := \text{template matching } t
\]
if success AND $S_b$-entry.request_type = "in" then
    $S_b$-entry.template := t
    $S_b$-entry.match := true
    sem := $S_b$-entry.semaphore
    Discard($S_b$) /* release read-write lock */
    V($S_b$,sem) /* unblock "in" request process */
    return
elseif success AND $S_b$-entry.request_type = "read"
    /* store tuple t in an appropriate group */
    i = tuple group determined by $S_a$
    Get($S_i$,read-write)
    Add tuple($S_i$,t)
    Discard($S_i$) /* release read-write lock */
    for each matching "read" request template do
        sem := $S_b$-entry.semaphore
        remove $S_b$-entry
        V($S_b$,sem)
        mark semaphore "sem" as unused in $S_b$-bitmap
    endfor
    Discard($S_b$) /* release read-write lock */
    return
endif

/* nobody is waiting for this tuple;
store tuple in an appropriate group */
i = tuple group determined by $S_a$
Get($S_i$,read-write)
Add tuple($S_i$,t)
Discard($S_i$) /* release read-write lock */
return

• in(s)

Using the association table $S_a$, one or more tuple group is searched for a matching tuple. If such a tuple is found, it is removed immediately and returned to the caller. Otherwise, the template $s$ is entered in an $S_b$ entry and the process waits on a semaphore. When the process wakes up, it reads and removes a matching tuple from the template in $S_b$ (that is put there by the out(t) request waking up the process).

The algorithm for in() is as follows:

Using $S_a$ decide on $S$, a set of tuple groups to search
for each $S_i$ in $S$ do
    Get($S_i$, read-write)
    success := tuple $t$ matching template $s$
    if success
        $t := S_i$-entry
        remove $S_i$-entry
        Discard($S_i$) /* release read-write lock */
        return($t$)
    endif
    Discard($S_i$) /* release read-write lock */
endfor
/* did not find any matching tuple; block waiting for an out(t) */
Get($S_b$, read-write)
sem := find a free semaphore using $S_b$-bitmap
$S_b$-entry := find a free entry in $S_b$
$S_b$-entry.template := $s$
$S_b$-entry.semaphore := sem
$S_b$-entry.request.type := "in"
$S_b$-entry.match := false
Discard($S_b$) /* release read-write */
P($S_a$, sem) /* block */
Get($S_b$, read-write)
t := $S_b$-entry.template
remove $S_b$-entry
mark semaphore "sem" as free in $S_b$-bitmap
Discard($S_b$) /* release read-write */
return($t$)

- read(s)

Using the association table $S_a$, one or more tuple group is searched for a matching tuple. If such a tuple is found, it is read and returned to the caller. Otherwise, the template $s$ is entered in $S_b$ and the process waits on a semaphore. When the process wakes up, it repeats the process of searching tuple groups looking for a matching tuple. The algorithm for read() is as follows:

Using $S_a$ decide on $S$, a set of tuple groups to search loop
    for each $S_i$ in $S$ do
Get($S_i$, read-only) /* we only need to read the group entries */
success := tuple $t$ matching template $s$
if success
  $t := S_i$-entry /* do not remove $t$ */
  Discard($S_i$) /* release read-only lock */
  return($t$)
endif
Discard($S_i$) /* release read-only lock */
endfor
/* did not find any matching tuple; block waiting for an out($t$) */
Get($S_b$, read-write)
sem := find a free semaphore using $S_b$-bitmap
$S_b$-entry := find a free entry in $S_b$
$S_b$-entry.template := $s$
$S_b$-entry.semaphore := sem
$S_b$-entry.request_type := "read"
$S_b$-entry.match := false
Discard($S_b$) /* release read-write */
P($S_b$, sem) /* block */
endloop

Other variations on the algorithm are possible. For example, it may be possible to divide $S_b$ into several groups instead of using one segment. Also, the read() operation can be implemented differently: The out() that unblocks read() operations writes into the templates the name of the group where tuple $t$ is stored ($S_i$). When read() wakes up as a result of the semaphore V operation done by the out(), it first searches $S_i$, and then (if necessary) the rest of the tuple groups. If no matching tuple is found, the read() operation modifies the template in $S_b$ to request another match and goes to sleep again. If found, it removes the template from $S_b$.

6 Process and Object Migration

As we mentioned earlier, Ra and DSMC are designed primarily to support the Clouds operating system. One of the design goals of Clouds is to provide distribution of data and computation transparent to the application. Objects in Clouds are passive and are the
encapsulation of code and data needed to implement the entry points in them. Threads are the only active entities in the system. A thread is a unit of activity from the users' perspective. Upon creation, a thread starts executing in an object. A thread enters an object by invoking an entry point in the object. It then executes the code in the entry point, and returns to the caller object. A thread in the course of its computation traverses the virtual spaces of the objects that it invokes. A thread is implemented as a collection of one or more processes. Each time a thread executes an RPC, a process is created on the remote node.

To efficiently exploit the resources available in a distributed system, Clouds should be able to migrate processes and objects among nodes. Through the DSMC primitives, Ra provides the operating system with mechanisms to migrate objects and processes.

As mentioned before, using DSMC, a segment can be activated and accessed on any node in the system (barring any protection limitations imposed by the operating system). An object is an O virtual space that consists of a collection of data and code segments with entry points, and the composition of the object is specified by a descriptor segment. An object, therefore, can be accessed on any node by activating the descriptor segment at the desired node. The DSMC primitives ensure that accesses to the object from different nodes are handled correctly.

Because objects are passive (i.e. consist of code and data only), process migration is performed separately from remote activation of O spaces. Note that a process consists of a P space and an isiba. To migrate a process, the operating system performs the following steps:

- The isiba is stopped (i.e. it is removed from the run queue).
- The kernel structure—Isiba control block (ICB)—that describes the isiba is copied to a new ICB on the remote node. The local ICB structure is returned to a local free list. (The ICB includes the name of the P space descriptor segment, as well as the
name of the segment that describes the object currently being invoked).

- The P space is deactivated (this can be done in the background; this step need not be completed before executing the following steps).
- The same P space is activated on the remote node. The O space in which the process was executing is activated, if need be.
- The new isiba is scheduled to run on the remote node.
- The process page faults on the required segments of the P and O space which will be fetched on demand.

The migration code is organized as follows. Each node has a process_migration system object that contains a migration routine (migrate) that migrates processes away from this node, and a migration server (migration_daemon) that accepts process migration requests to the local node. Each migration_daemon has associated with it a communication segment and a semaphore. The communication segment is arranged as a circular list of ICB slots. The migration system has a simple name service mechanism that maintains tables containing the system names for the communication segment and semaphore for each <host,migration_daemon> pair. The semaphores used by the migration system belong to a well-known semaphore segment (semaphore_seg). To migrate a process, migrate is called giving the name of the process to migrate and destination host id:

\[
migrate(SysName \text{ process, HostID host})
\]
\[
ICB = stop_process(process)
\]
\[
comm\_seg = lookup\_comm\_segment(host, migration\_daemon)
\]
\[
semaphore = lookup\_semaphore(host, migration\_daemon)
\]
\[
Get(comm\_seg, read-write)
\]
\[
copy ICB to next free slot in comm\_seg
\]
\[
Discard(comm\_seg)
\]
\[
V(semaphore\_seg, semaphore)
\]
\[
deactivate(ICB.P\_space)
\]
return ICB to free list
return

The migration_daemon at a node blocks on its semaphore. On unblocking, it reads the ICB from the next slot in its communication segment. This information is copied to a new ICB, and the P and O spaces are activated. Finally, a kernel routine is called to make the ICB runnable.

migration_daemon()
    comm_seg = lookup_comm_segment(my_host(), migration_daemon)
    semaphore= lookup_semaphore(my_host(), migration_daemon)

    loop
        allocate(ICB)
        P(semaphore_seg, semaphore)  /* block waiting for request */
        Get(comm_seg, read-write)
        ICB := comm_seg.next_slot()
        Discard(comm_seg)

        activate(ICB.P_space)
        if not active(ICB.0_space)
            activate(ICB.0_space)
        schedule(ICB)
    end loop

Except for copying the ICB structure (around 130 bytes in the current implementation), all required portions of the P space are fetched on demand only. Zayas [Zay87] has shown that, when migrating processes, using on-demand fetching has considerable performance advantages over copying all of the process’ data at the time of migration.

At any point of time, a segment is owned by one node only. Segment migration is the process of passing the ownership of a segment from one node to another, and is accomplished by copying the whole segment from the stable storage of the previous owner to the stable storage of the new owner. To permanently move an object to another node, the segments that make up the object need to be migrated. Note that, in general, the segment that

\[\text{This requires deactivating the segment first and updating/invalidating location caches, if any.}\]
describes an object, as well as the segments that make up the object, can be owned by different nodes. Therefore, if a decision is made to migrate an object to one node, several nodes may have to be involved in transferring the ownerships of segments to the destination node.

Process and object migration uses the DSMC and Ra primitives without having to add additional mechanisms or communication protocols, and without a need for modifying the kernel. This feature is in contrast to other implementations of process migration such as in V [TLC85], Accent [Zay87], and DEMOS/MP [FM83] systems.

7 Two-phase Commit

In distributed systems, there is often a need for a mechanism that ensures that a single logical operation is consistently carried out at multiple nodes. In our system, we want to be able to modify segments from different owners, and then save all changes at all or none of the owners. Such a mechanism can be used to commit or abort transactions [BHG87]. The two-phase commit protocol [BHG87] can be used to implement such a mechanism.

In the two-phase commit protocol, one node initiates the algorithm and acts as a co-ordinator. During the first phase, the coordinator sends vote requests to all participants. Upon receiving a vote request, each participant replies with a yes or no answer. During the second phase, the coordinator collects all votes from the participants. If all votes are yes, the coordinator decides to commit and sends commit messages to all participants. Otherwise, it decides to abort and sends abort messages instead. Each participant that votes yes waits for the coordinator’s decision, and acts accordingly.

In our system, a commit or an abort request can be issued for a series of accesses to segments with the recoverable attribute. Access to such segments is handled by the operating system as follows: When a process requests access to a recoverable segment (e.g. by faulting on a location in a recoverable segment), a get() request is issued by the operating system.
to the owner of the segment. The segment is then copied to a new private segment. The
requesting process resumes execution with all accesses going to the private segment. When
the process issues a commit request, the following two-phase protocol is executed by the
coordinator:

- **Phase I:**

  ```
  for each recoverable segment touched by the process do
  segment_migrate(private segment, owner of recoverable seg)
  "inform" all owners to flush to stable storage all
  the migrated segments
  endfor
  ```

- **Phase II:**

  ```
  "collect" answers
  if all answers are yes then
  "inform" all owners to replace the old seg with the
  new (private) segment
  else
  "inform" all owners to
  abort by destroying the private seg
  endif
  ```

Informing and collecting votes, and segment-migration are implemented using the data
transfer and the semaphore primitives provided by the DSMC. The coordinator and the
participants exchange information by writing and reading common memory segments, and
use the locking and semaphore primitives to synchronize access to these segments.

There are points in the protocol where a node needs to write to stable storage a record of
its action before proceeding [BHG87]. For example, the coordinator has to write its decision
to stable storage before telling the participants to abort or commit. This write is needed
in case of a coordinator failure; the coordinator needs some means of knowing what it was
doing before it failed. Likewise, each participant needs to write a record to stable storage
before informing the coordinator of its vote. Therefore, each diskless protocol participant
has to write its vote in a segment, discard the segment to a node with stable storage and wait for it to be written out to disk, before continuing with the protocol.

8 Distributed Game

Certain applications may not require the strong consistency conditions imposed by the read-write, read-only, and none modes of the DSMC. For example, in a distributed game in which the state of each player is visually seen on a CRT screen, it may often be sufficient to show the “most recent” state and not necessarily the “correct” state. For such applications a weaker form of consistency semantics would suffice. The weak-read mode of the DSMC has this intent.

In this section we describe the implementation of a multiplayer distributed game using the DSMC primitives. The game is similar to the Amaze game described by Berglund et. al [BC85], where each player maneuvers a monster through a maze. The objective of the game is to shoot the monsters controlled by the other players. Each player uses a workstation that displays the maze on a graphics screen. The keyboard is used to control monsters and the firing of missiles.

Amaze is implemented using several processes (including a keyboard process, a timer process, and a manager process) at each workstation that communicate using messages[BC85]. In addition, there is a status reader process per participant in the game. The manager process blocks waiting for messages from one of its local processes and updates the display screen periodically. Each status reader process blocks on reception of status update messages from a remote workstation. When a status reader process receives such a message, it sends a message to its manager process to update the game state. The implementation attempts to minimize update messages among the different workstations by sending messages only at major events (such as the firing of a missile or a monster change of direction). Periodically, update messages are sent to synchronize the different copies of the game.
An implementation of the game in our environment uses a similar organization of processes. However, processes communicate through shared memory and semaphores instead of messages, and each node has only one status reader process instead of one per participant. The status reader process blocks on a semaphore waiting for game status changes. The allocation of the semaphore is made when a player joins the game. A shared segment (map) contains a map of available and allocated semaphores. A workstation joining the game inspects this segment to acquire a semaphore and to find out the other players in the game.

A game status change is communicated to other players by (1) acquiring a read-write lock to a shared segment (comm_seg) which is used to communicate the state of the game and writing a record of the change in the segment, then (2) unblocking each status reader process by issuing V operations on the corresponding semaphores. An unblocked status reader process acquires a copy of the shared segment using weak-read mode and reads in the latest change of state. Note that the shared segment is maintained as a circular list of status changes records so a change record is eventually removed when the list wraps around.

The status_reader_process is implemented as follows:

```c
status_reader_process
  join_game(my_semaphore, current_players) /* initialize */
  loop
    P(sem_seg,my_semaphore) /* wait for status change */
    Get(comm_seg,weak_read)
    state := read_next_record(comm_seg)
    update local state
    V(sem_seg,local_manager_process)
  endloop
```

The join_game routine inspects the shared segment map, acquires a free semaphore, initializes the local list current_players to include the semaphore numbers of other status readers processes, and calls communicate_state_change to announce that a new player has joined the game. Join_game is implemented as follows:

```c
join_game(int my_semaphore, list current_players)
  Get(map, read-write)
  my_semaphore := allocate_semaphore(map)
  current_players := read_current_players(map)
```
Discard(map)
communicate_state_change(<join,my_semaphore>)
return

Communicate_state_change routine is implemented as follows:

communicate_state_change(record state)
    Get(comm_seg,read_write)
    write_next_record(comm_seg,state)
    Discard(comm_seg)
    for each sem in current_players do
        V(sem_seg,sem)
    endfor
    return

9 Comparison of DSM vs. RPC

The earlier sections illustrated how different diverse distributed applications can be implemented using only DSM primitives. These applications can just as well be implemented using only RPC. It is our belief that a combination of RPC and DSM will be most effective for coding many distributed applications. It would be instructive to implement these applications using only RPC, only DSM, and a combination of the two, and compare their performance. We are in the process of doing such a study.

In the meanwhile, we have conducted simulation studies that compare using RPC and DSM for implementing remote object invocation under synthetic workloads (see [RAK88] for more details). In the simulation, the DSMC is treated as a software module as described in §4. The performance study shows that:

- Over a range of object invocation locality, DSMC has significant advantages over RPC.
- Use of distributed shared memory achieves automatic distribution of processing load, by performing the invocation locally instead of on the owner nodes.
• Instead of addressing memory coherency and synchronization separately, applications that can be modeled as readers/writers problems benefit considerably when locking and segment access primitives are combined.

As mentioned before, the DSMC primitives are provided to the operating system as a set of mechanisms for managing memory in the network. Policy decisions, such as when to use RPC and when to use the DSMC primitives, are in the operating system. For example, during periods of high concurrent access to an object, the operating system may use the DSMC primitives to locate the object at one node and then use RPCs to invoke the object thereby reducing data movement across the network.

10 Conclusions and Future Work

We have shown how the concept of distributed shared memory is a viable mechanism for programming distributed applications. We proposed a simple set of primitives, described its implementation, and illustrated its use in four different applications. The highlights of our work that distinguishes it from other related work are the following:

• In an environment consisting of diskless workstations and a few data servers, the twin requirements of remote paging and data sharing (and hence interprocess communication) are combined into one powerful set of mechanisms.

• We have integrated distributed shared memory into the design of the Ra kernel thereby simplifying network-wide memory management.

• We have provided shared memory as a viable alternative for remote object invocation.

We plan to design the DSMC as a hardware module that, along with an ethernet interface, implements the distributed shared memory primitives and offloads this work from the main CPU.
Acknowledgment

We thank Christopher Wilkenloh for implementing the TAL protocol.
References


Figure 1: Organization of DSMC implementation under Ra
Figure 2: Organization of DSMC implementation on Unix
An Implementation of Distributed Shared Memory*

Umakishore Ramachandran  
School of ICS  
Georgia Institute of Technology  
Atlanta, Ga 30332  
Phone: (404) 894-5136  
E-mail: rama@gatech.edu

M. Yousef A. Khalidi  
Sun Microsystems  
2550 Garcia Avenue  
Mountain View, Ca 94043  
Phone: (415) 336-5392  
E-mail: yak@sun.com

Abstract

Shared memory is a simple yet powerful paradigm for structuring systems. Recently, there has been an interest in extending this paradigm to non-shared memory architectures as well. For example, the virtual address spaces for all objects in a distributed object-based system could be viewed as constituting a global distributed shared memory. We propose a set of primitives for managing distributed shared memory. We present an implementation of these primitives in the context of an object-based operating system as well as on top of Unix.

1 Introduction

Programming with shared memory is well-understood and despite the interest in distributed and parallel systems for reasons of availability, fault-tolerance, and increased computational power, the style of programming these systems has not changed drastically. Even in non-shared memory architectures researchers have proposed a style that presents to the programmers an abstraction of a logical shared memory [19, 14, 8, 23]. Other researchers have proposed algorithms for maintaining the consistency of such a logically shared memory in non-shared memory architectures [17, 18, 21]. The abstraction for supporting the notion of shared memory on a non-shared memory (distributed) architecture is referred to as distributed shared memory (DSM) in this paper.

A second motivation for DSM is the current trend in structuring distributed systems using a collection of diskless computational servers, namely workstations, and a few data servers or file servers. In such an environment the code and data for program execution has to be paged-in from the data server. There are two issues here: The first one is a scheduling decision of 'where' to execute the program, one that is best left to a higher level policy making entity. The second one is the chore of bringing in the required data and code, i.e.,

*This work has been funded in part by NSF grants CCR-8619886 and MIPS-8809288.
remote paging. If sharing is coupled with this second issue, then we see that DSM presents itself as a natural facility for combining the two.

Several other researchers have proposed software architectures based on the shared memory paradigm, in different settings:

- Li [18] presents a variation of the Berkeley protocol for multiprocessor cache consistency [15] as a solution to maintain the consistency of distributed shared memory. Using Li’s scheme, the entire memory in the distributed system is considered potentially sharable for both reads and writes. The consistency protocol maintains the coherency of memory even when accessed by processes running on different nodes.

- In a speech recognition application, Bisiani and Forin [8] use data structures that are shared by multiple language modules that are distributed on heterogeneous machines. They show that communication through shared memory is a viable alternative to message-passing even when the environment involves cooperation between multilingual program modules and heterogeneous machines.

- Processes in the programming language Linda [9, 13] communicate via a globally-shared collection of ordered tuples.

- A logically shared bulletin-board is proposed by Birman, et al. [7] for structuring asynchronous interactions between processes in distributed systems.

- By integrating the mechanisms for virtual memory management and local interprocess communication, Mach [25] achieves efficient implementation of local interprocess communication. Currently, researchers at CMU are investigating the duality of shared memory and message passing in the context of network communication as well [30].

- Zayas [31] achieves substantial reduction in the cost of process migration by using copy-on-write techniques [24] and on-demand fetches during remote execution.

- Cheriton [10] advocates problem-oriented shared memory as the basic concept for structuring distributed systems.

- Emerald [14] is a distributed object-based language and system with support for object mobility.

The purpose of this paper is to present a set of mechanisms for DSM and an implementation of these mechanisms. All the resources of the system are viewed as potentially shared objects. The name space of these objects constitute a distributed shared memory. The objects are composed of segments, where a segment is a logical entity that has attributes such as read-only, and read-write. There is a concept of ownership whereby the node where a segment is created (the owner node) is responsible for guaranteeing the consistency of the segment. The distributed shared memory controller (DSMC) to be described next is the entity that provides the mechanisms for managing these segments.

2 Distributed Shared Memory Controller

The basic operations provided by the DSMC are get, and discard. The get operation is used to fetch a segment from its owner, while discard is used to return a segment to its owner. The DSMC provides synchronization primitives as separate operations (P and V semaphore operations), or as combined access and lock operations using the get and discard primitives.
Using the get primitive a segment may be acquired in one of four modes: read-only, read-write, weak-read, or none. Read-only mode signifies non-exclusive access but guarantees that the segment will not change until the node explicitly discards the segment. Read-write mode signifies exclusive access (for the node) with a guarantee that the segment will not be thrown away until the node explicitly discards the segment. When a get primitive is issued with mode read-only or read-write the local DSMC sends a request to the owner DSMC and suspends the requesting process until the segment is received. The segment is kept until an explicit discard is issued. Multiple copies of the segment may be held by several readers at the same time (mode read-only) but only one writer (mode read-write) may have access to the segment at a time. The owner node keeps a count of the number of requesters that have a copy of the segment in read-only mode.

Weak-read mode signifies non-exclusive access with no guarantee whether the segment will change or not. The owner DSMC immediately honors a weak-read requests by sending a copy of the segment to the requesting DSMC. None mode signifies exclusive access with no guarantee whether the segment will be thrown away or not. When a get primitive is issued with mode none the local DSMC sends a request to the owner DSMC and suspends the requesting process until the segment is received. None mode requests are enqueued in the appropriate segment queue, if the segment is currently held in either read-only or read-write modes. If the segment is available at the owner DSMC, it responds by sending the segment to the requesting DSMC. The requesting DSMC becomes the keeper of the segment and the owner remembers the current keeper. If the segment is held in another node in mode none, then the owner DSMC instructs the current keeper to forward the segment to the requesting DSMC. A segment held in mode none at a keeper node may be returned to its owner by issuing a discard primitive, or it may be taken away by its owner when the keeper DSMC is instructed to forward the segment to another node.

The DSMC also provides the semaphore operations P and V that act on semaphores that are contained in semaphore segments (see §5).

3 Clouds

While the mechanisms provided by DSMC are general, we describe an implementation of these mechanisms in the context of Clouds, an object-based distributed operating system. Therefore, a brief description of Clouds is appropriate.

Clouds, being developed at Georgia Tech [5], is intended to provide a unified environment over distributed hardware. Location independence for data as well as processing, atomicity of distributed computation, and fault-tolerance are some of the research goals of Clouds. Objects and threads are the basic building blocks of Clouds. Objects are passive entities and specify a distinct and disjoint piece of the global virtual address space that spans the entire network. An object is the encapsulation of the code and data needed to implement the entry points in the object. Thus a Clouds object can be considered syntactically equivalent to an abstract data type in the programming language parlance. Access to entry points in the object are accomplished through a capability mechanism in software.

Threads are the only active entities in the system. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object. A thread enters an object by invoking an entry point in the object. It then executes the code in the entry point, and returns to the caller object. Binding the object invocations to the entry points in the object takes place at execution time. Figure 1 shows the model of computation in Clouds. A thread in the course of its computation traverses the virtual address spaces of the objects that it invokes.

In a distributed object-based system, the virtual address spaces of all objects can be
viewed as constituting a global distributed shared memory. Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-management mechanism.

For remote object invocation there are two choices: The first choice is to perform the computation at the node where the object resides, referred to as remote procedure call. The second choice is to make the invocation appear local by bringing in the segments required for the invocation. While we have to support the former for immovable objects, such as an object that reads disk blocks, we believe that the latter may be a better choice for movable objects. There are two reasons to support this belief:

- the principle of locality [12] that suggests an invocation or other invocations in the same object may be repeated
- the reduction in computational overhead due to the elimination of slave process management to support remote invocation at the node where the object resides [16, 22].

4 The Structure of Clouds

4.1 Ra Kernel

Ra [4, 6] is an operating system kernel designed to be the nucleus of Clouds operating system [5]. It is currently implemented on the Sun-3 architecture. Ra defines and manages three primitive abstractions: segment, virtual space, and isiba. Segments serve as containers of data and may be viewed as uninterpreted sequences of bytes. The contents of a segment may only be accessed when the segment is attached to a range of virtual addresses. Segments persist until explicitly destroyed. Each segment resides in a partition that is responsible for providing backing store for the segment. A partition is an entity that realizes, maintains, and manipulates segments (see §4.2).

Virtual spaces abstract the notion of an addressing domain. A Ra virtual space is a monotonically increasing range of virtual addresses with possible "holes" in the range. A virtual space has a descriptor segment associated with it that contains a collection of windows. Each window is a data structure that maps a contiguous piece of the virtual space to a segment. Figure 2 shows the relationship between the Ra virtual space, the windows, and the segments. The segmentation scheme in the Chorus system [2] has some similarity to the Ra virtual space.

![Figure 1: Model of Computation](image-url)
Virtual Space defined by descriptor

Figure 2: Ra Virtual Space
Ra isibas are an abstraction of the fundamental notion of computation or activity and can be thought of as light-weight processes. Isibas may be used as daemons within the kernel or they may be associated with a Ra virtual space to implement a user process. A Clouds thread can potentially span machine boundaries and is implemented as a collection of processes.

A Ra virtual space is a software abstraction not to be confused with the virtual address space provided by the machine architecture. The latter is assumed to be composed of three distinct regions that are called O, P, and K spaces for object, process, and kernel, respectively. Note that such a distinction may not exist in a given machine architecture. In that case the division is enforced based on address range 'high' and 'low' water marks.

The kernel is mapped in the K space. A process consists of an isiba and a Ra virtual space that contains the process stack for invocations. Note that a process' virtual space does not contain any code. A process' virtual space is mapped into the P space and unmapped on context switch. An object is a Ra virtual space that consists of code and data segments. The code segment of an object's virtual space has entry points that can be invoked by user processes. The object in which a process is currently executing is mapped into the O space. System objects, which we discuss in §4.2, are mapped into the K space, but may be installed and removed dynamically.

4.2 System Objects

System objects are trusted software modules that are loaded dynamically in the K space. System objects encapsulate necessary and/or useful operating system services and resource managers that have direct access to the Ra kernel, but are nonetheless outside the kernel. They implement and encapsulate policy as the kernel itself does not make any policy decisions. System objects serve as intermediaries between the user objects and the kernel, and they provide system services to user objects. Examples of system objects include resource managers, user-level object support, device drivers, and partitions. Of particular concern to this paper is the partition system objects.

The Ra kernel runs on machines that provide support for virtual memory. The Ra kernel is responsible for mapping segments into virtual memory using the memory management hardware provided by the underlying architecture. The size of a segment is a multiple of the physical page size. Ra assumes the existence of partitions that are responsible for storing segments.

Segments are maintained by partitions, and a segment is said to be controlled by a partition. Several operations are possible on segments via their controlling partition: Segments may be created and destroyed. The page-in and page-out operations on segments allow the partition to cooperate with virtual memory management in order to access the contents of a segment and to update its representation on secondary storage when necessary. Finally, segments may be activated and deactivated. Activating a segment prepares the partition for further activity relating to the segment, while deactivating a segment informs the partition that further access to the segment is unlikely in the near future. The activate and deactivate operations are similar to open and close file operations in conventional systems.

Therefore, each partition provides, at least, the following calls for use by Ra: activate/deactivate segment, create/destroy segment, and page-in/page-out portions of segments. Ra services segment requests from other system objects, such as to map a segment into a virtual space. In addition, Ra fields page faults, determines the virtual space and in turn the segment where the fault occurred, and calls the appropriate fault handler. When Ra is instructed to service a segment request (e.g. to map a segment into a virtual space), it invokes the appropriate partition to fetch the segment into physical memory. Ra then manipulates the memory management hardware to map the physical pages appropriately.
The Ra architecture assigns to the kernel the task of mapping a segment onto the memory management hardware, and hides the details of storing the segment on external storage in the partition system objects. The fact that the segment is stored on a local disk or on a remote node is hidden from the kernel.

5 Implementation on Ra

5.1 Overview

Figure 3 shows the organization of the DSMC implementation (roughly 3500 lines of C++ code [26]) on Ra. The boxes in the Figure denote system objects. The DSMC cooperates with remote DSMC's to implement the distributed shared memory primitives. The DSM partition is a Ra partition that provides the kernel with the ability to create/destroy and activate/deactivate segments, page-in/page-out portions of segments, and semaphore P/V operations. The DSM partition decides if a segment is owned by the local node or a remote node. It uses the Disk partition to access local segments, and uses the DSMC to access remote segments. The Disk partition maintains segments owned by the local node on the local stable storage (if any).

The DSMC algorithms require simple reliable request/response messages (possibly with message forwarding). In our implementation, we use the transaction abstract layer that is built on the Ra Transaction Support Protocol (TAL/RaTP) [29]. TAL/RaTP protocol is similar to other transaction-oriented protocols such as VMTP [11]. However, it is much simpler than VMTP since it is tailored to our requirements.
5.2 Handling Local Requests

5.2.1 DSM Partition

DSM partition provides the minimum set of partition operations plus the semaphore operations. The DSM partition handles segment requests from the Ra kernel or from other system objects (Figure 3). An example of a system object that uses the DSM partition is the user-level object handler that is responsible for implementing object invocation and servicing user-visible segment operations. Such user-visible operations may include lock/unlock and P/V operations.

The DSM partition maintains the status of cached (local and remote) segments on the local node in a table called dtable (Figure 4). Each dtable entry maintains information about a block of a segment (where a block is a multiple of the physical page size). In the current implementation, a block is equal to the physical page size on the Sun-3 (8K bytes). Each valid entry in the dtable is doubly linked on a hash list. The table is hashed by the segment name, and searched with the key <segment name, block#>. All free entries are linked on a free list. In addition, an active segment table (ast) contains an entry per active segment on the local node.

Each dtable entry represents one segment block and includes the following fields:

- **segment, block_number** — These two fields identify the segment block represented by this entry.
- **wait_lock** — A lock that is used to synchronize access to this entry.
- **phys_frame** — An array of physical frame numbers that contain this block (in the current implementation, the cardinality of the array is one).
• pending — A flag indicating that a read from disk is in progress, or a get with mode none has been issued to the owner DSMC.

• Mode — This field indicates the current mode of the block.

• readers — A field that indicates the number of requesters that have this block in read-only mode.

• owner_flag — A flag that indicates if the local node is the owner of the segment.

• keeper_owner — This field gives the current keeper of the segment (if owner_flag is true), or the owner of the segment (if owner_flag is false).

• squeue — A list used by the owner DSMC to queue read-only and read-write requests for this block from remote DSMCs.

The DSM partition operations can be classified into three groups:

• Control Operations:
  - activate(segment)
  - deactivate(segment)
  - create(segment)
  - destroy(segment)

The control operations search the ast for an entry describing the segment. If an entry is not found, the location system object is consulted for the location of the segment.¹ If the segment is available on a local disk partition, the corresponding control operation is invoked on the disk partition. Otherwise, the segment is owned by a remote node, and a msg_control message is sent to the DSM partition on the remote node. Note that locating the segment is the responsibility of the location system object, and that the DSMC is not involved in handling any of the control operations.

• Data Transfer Operations:
  - page_in(segment, block, physical page)
  - page_in(segment, block, physical page, mode)
  - page_out(segment, block)

The page_in operation activates the segment, if necessary. The page_in operation searches the dtable for an entry describing <segment, block>. If no such entry is found, an entry is created. For segments owned by the local node, the page_in operation on the disk partition is invoked. For remote segments, the DSM partition translates the page_in requests to the DSMC get operations with the specified mode. If no mode is indicated in the page_in call, mode none is assumed. The page_out operations locates the dtable entry describing <segment, block>, and invokes the page_out call on the disk partition if the segment is local, or calls the DSMC discard operation, if the segment is remote.

• Synchronization Operations:
  - P(segment, semaphore_num)
  - V(segment, semaphore_num)
Semaphores are stored in *semaphore segments*. Semaphore segments have the format shown in Figure 5. Each semaphore segment consists of three parts: a descriptor structure, *n* semaphore structures, and *m* block structures. Descriptor contains the number of semaphore and block structures, a bitmap of free/used semaphore structures, and a pointer to a free list of block structures. Each semaphore structure describes a semaphore, and includes a counter and a pointer to a doubly-linked list of block structures. Each block structure describes a process waiting for a semaphore. Each block contains the name of the waiting process, the host name where the process is blocked, and a pointer to the *isiba control block* (ICB) on the host where the process is blocked. Note that the host name and ICB pointers are hints to the location of the blocked process because processes can migrate from one node to another. The process name field is an absolute pointer to the process and can be used by the locator system object to find the process if necessary. Processes (as well as other entities in Ra) have unique network-wide names.

Semaphore segments can be attached to a range of virtual addresses like any other segment in Ra. Therefore, they can be initialized and manipulated directly. The DSM partition maintains a table (called *semtable*) that describes each semaphore in use. *Semtable* acts as a cache of the active semaphores that are in the semaphore segments. *Semtable* is organized in a similar fashion as the dtable, but it is searched using the key <segment, semaphore number>. Each semtable entry caches exactly one semaphore from a semaphore segment. The structure of each semtable entry is the same as the semaphore structure in semaphore segments (see Figure 5). A pool of in-memory block structures are used to cache contents of block structures from semaphore segments.

---

1. Given the name of a segment, the location system object returns the location of the segment owner. A simple location system object broadcasts a search request for each location operation (see [3, 1] for more sophisticated location algorithms).
<table>
<thead>
<tr>
<th>Type</th>
<th>From</th>
<th>To</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg_control</td>
<td>DSM part.</td>
<td>DSM part.</td>
<td>used for control operations</td>
</tr>
<tr>
<td>msg_reply</td>
<td>DSM part.</td>
<td>DSM part.</td>
<td>reply to msg_control</td>
</tr>
<tr>
<td>msg_get</td>
<td>keeper DSMC</td>
<td>owner DSMC</td>
<td>fetches a segment block</td>
</tr>
<tr>
<td>msg_discard</td>
<td>keeper DSMC</td>
<td>owner DSMC</td>
<td>returns a segment block</td>
</tr>
<tr>
<td>msg_forward</td>
<td>owner DSMC</td>
<td>keeper DSMC</td>
<td>forwards a segment block</td>
</tr>
<tr>
<td>msg_segment</td>
<td>any DSMC</td>
<td>any DSMC</td>
<td>delivers requested segment block</td>
</tr>
<tr>
<td>msg_P</td>
<td>keeper DSMC</td>
<td>owner DSMC</td>
<td>semaphore P operation</td>
</tr>
<tr>
<td>msg_V</td>
<td>keeper DSMC</td>
<td>owner DSMC</td>
<td>semaphore V operation</td>
</tr>
<tr>
<td>msg_unblock</td>
<td>owner DSMC</td>
<td>keeper DSMC</td>
<td>continues suspended process</td>
</tr>
<tr>
<td>msg_error</td>
<td>any</td>
<td>any</td>
<td>indicates an exception</td>
</tr>
</tbody>
</table>

Table 1: Summary of TAL messages

Operations on a semaphore that belongs to a local segment are performed locally by reading the segment from the disk partition, initializing a semtable entry, and then performing the operations on the semtable entry. In the current implementation, all semaphore operations are performed at the node that owns the semaphore segment. Therefore, operations on semaphores that belong to remote segments are translated into DSMC P and V operations (see §5.5).

5.2.2 DSMC

The DSMC provides the following four operations for use by the DSM partition:

get(dtable_index, mode)
discard(dtable_index)
P(semtable.index)
V(semtable.index)

In order to implement these primitives, each DSMC uses the TAL/RaTP messages listed in Table 1 to communicate with other DSMCs (see §5.3.2).

5.3 Handling Remote Requests

5.3.1 DSM Partition

DSM partitions exchange msg_control and msg_reply messages to implement the activate, deactivate, create, and destroy control operations, as described in §5.2.1. In addition, each DSM partition services requests from its local DSMC to activate local segments, to read a block of a local segment, and to initialize a semtable entry from a local semaphore segment (see §5.3.2).

5.3.2 DSMC

The DSMC may receive several messages from remote DSMCs. We describe the DSMC action for each message received:
- `msg_get(segment, block, mode)`
  If there is no ast entry for this segment, the DSMC asks the DSM partition to activate
  the segment and to read the required block into memory. The DSMC examines the
  dtable entry describing the required block. Depending on the information contained
  in the dtable, the DSMC performs one of the following actions:

  - If the segment mode does not conflict with the requested mode and the block
    is available locally, send a `msg_segment` message that includes the block to the
    requesting DSMC
  - If the segment mode is `none`, the requested mode is also `none` and the segment is
    held at a remote node, then send a `msg_forward` message to the remote DSMC,
    instructing it to forward the block to the requester
  - If the segment mode and requested mode conflict (e.g. segment is held in
    read-only mode and requested mode is read-write), queue request until seg-
    ment is available
  - If the segment does not exist at this node, then send a `msg_error` message to
    the requesting DSMC.

- `msg_discard(segment, block)`
  The DSMC updates the dtable entry describing the discarded block. If there exist
  any pending `get` requests for this block that now can be satisfied, they are serviced
  by sending `msg_segment` messages to the requesting DSMCs.

- `msg_segment(segment, block)`
  The DSMC receives a `msg_segment` message as a response for a `msg_get` message.
  The DSMC locates the dtable entry describing the block, and resumes the suspended
  processes that are awaiting the arrival of the block.

- `msg_forward(segment, block, destination host)`
  The DSMC informs the DSM partition that this block is no longer available, and then
  issues a `msg_segment` message containing the required block to the destination host.

- `msg_P(segment, semaphore_num)`
  If there is no semtable entry for the required semaphore, the DSMC requests its
  local DSM partition to initialize an entry. The DSM partition may have to acti-
  vate the required semaphore segment and then read the information of semaphore
  semaphore_num into the new semtable entry. The DSMC decrements the semaphore
  count, and responds with a `msg_unblock` message if the count is greater than or equal
  to zero. Otherwise, it links to the semtable entry a new block structure that describes
  the requesting process.

- `msg_V(segment, semaphore_num)`
  If there is no semtable entry for the required semaphore, the DSMC requests its local
  DSM partition to initialize the entry. The DSMC increments the semaphore count.
  If the count is less than or equal to zero, the DSMC unlinks the first block structure
  from the semtable entry and sends a `msg_unblock` to resume the process described
  by the unlinked block structure.

- `msg_unblock(segment, semaphore_num, ptr to ICB)`
  The DSMC resumes the execution of the waiting process identified by the `msg_unblock`
  message.
\textbf{msg\_error}(segment, block, error\_type)

An error indication may be received if a request cannot be satisfied. The error\_type field gives the reason for the failure of the request. Upon receiving an \textbf{msg\_error} message as a response for a request, the DSMC returns an error indication to the original requester.

5.4 Table Management

As mentioned in §5.2.1, the DSM partition is responsible for maintaining \texttt{ast}, \texttt{semtable}, and \texttt{dtable} structures. The size of \texttt{ast} is equal to the maximum number of active segments at any point of time, and \texttt{ast} entries can be reclaimed when the segments they represent are deactivated. \texttt{Semtable} acts as a cache of the information contained in semaphore segments, and its size is equal to the expected number of semaphores in use. To reclaim a \texttt{semtable} entry, the contents of the entry has to be written back into its semaphore segment. The size of the \texttt{dtable} depends on whether or not the node acts as an owner of segments. For nodes that act only as keepers, the size of the \texttt{dtable} is less than or equal to the number of physical pages at the node. For a node that acts as an owner, the size of the \texttt{dtable} is determined by the number of nodes serviced and the size of their physical memories. A \texttt{dtable} entry is reclaimed when the block it represents is paged-out. If the entry represents a block that is cached at a keeper node, the entry can be reclaimed when the block is returned to the owner.

5.5 Performing Semaphore Operations Locally

As described in §5.2.1, all semaphore operations in the current implementation are performed at the node that owns the semaphore segment. To exploit synchronization locality (e.g. when all processes using the same semaphore are at the same node), it should be possible to perform the semaphore operations at the local node without the intervention of the owner node on each operation.

Because semaphores reside in segments, it is possible to fetch the semaphore segment from its owner, read its contents into the local \texttt{semtable}, and perform the operations locally. When a decision is made to move the semaphore segment from its current node, the DSM partition must ensure that the contents of the segment is up to date by flushing any \texttt{semtable} entries that belong to the segment prior to sending the segment to another node. Processes blocked on a semaphore need not be migrated when the semaphore segment is moved to another node, because the semaphore segment contains the name of the host where the process is blocked (Figure 5).

The following simple modifications to the DSMC \texttt{P} and \texttt{V} primitives are required: Instead of sending all semaphore operations to the owner DSMC, a check is made to see if the semaphore segment is cached locally. If it is, the operation is performed locally and the owner DSMC is not contacted. Otherwise, the semaphore operation is sent to the owner DSMC. When the owner DSMC receives a semaphore operation message (\texttt{msg\_P} or \texttt{msg\_V}), it checks to see whether the required semaphore segment is available locally, or is cached at a remote keeper. If the semaphore segment is available locally, the semaphore operation is performed as before at the owner node. Otherwise, the semaphore operation is forwarded to the current keeper of the semaphore segment. Note that the owner DSMC maintains at all times the location of the current keeper of each segment, and therefore can easily forward semaphore operations on segments that are cached at remote keepers.

The semaphore mechanisms presented in this section do not address the issue of where (or when) to move semaphore segments. Instead, they perform the \texttt{P} and \texttt{V} semaphore operations at the \textit{current} location of the semaphore segment. Other system objects are
responsible for deciding on where to place semaphore segments in the distributed system, and when to move them to other nodes.

5.6 Fault Tolerance

The DSMC implementation assumes the existence of a reliable transport protocol underneath. Any failures in the network result in an 'error' indication being propagated to the system object that made the request to the DSMC. Recovering from such failures could possibly involve reconstructing the segments at a different node. Failure handling clearly involves policy issues, best left to appropriate system objects. The DSM layer concerns itself only with segment transport and gives the necessary error indication to higher level system objects for appropriate corrective action. In our view, fault-tolerance has to be addressed in distributed systems regardless of whether RPC or DSM is used as a mechanism for remote invocation, and we plan to do this as part of our future research.

6 Implementation on Unix

The DSMC and the DSM partition have been implemented on top of Unix as well. This implementation serves three purposes:

1. The Unix environment makes it easy to test and verify the DSMC and TAL/RaTP protocols.

2. The Unix file system is available for use as permanent store for segments. Ra executes on diskless Sun-3 workstations with backing store provided by Unix machines.

3. The strength of Unix is the rich program development environment that it provides. The strength of Clouds is the transparent management of distributed data and computation. Providing inter-operability between Unix and Clouds is one of our design goals. DSM implementation on Unix and Ra serves this purpose. System and user objects are developed on Unix and demand-paged to Ra via the DSM mechanisms.

The organization of the DSMC implementation on Unix is shown in Figure 6. TAL/RaTP runs as a user process that uses SUN's Network Interface Tap (NIT) [27] to receive packets from the net and to route them among a set of clients and servers. The DSMC code is linked-in with the server code that uses the Unix file system to store segments and service requests from Ra DSM partitions. The DSM code is also linked-in with client code that is used to test the DSM system.

Most of the DSMC and DSM partition code that runs on Ra are re-used for the Unix implementation, and the operating system dependencies are isolated in a few C++ classes. To enable more than one Unix process to share the DSM tables, we use Unix System V shared memory regions [27]. In addition, Unix System V semaphores are used to synchronize access to the tables. In our initial implementation, we also used System V semaphores to synchronize the TAL/RaTP process and its client processes. A client process requested TAL/RaTP services by writing in a shared region of memory and blocking on a semaphore. TAL/RaTP eventually resumed the process by issuing a V operation on the semaphore.

In the current implementation, however, we switched to using Unix 4.2 BSD socket IPC primitives instead of System V semaphores because of the poor performance of the initial implementation (see §7). The TAL/RaTP process communicates with its client processes through shared memory and sockets. Processes communicate their requests to TAL/RaTP via socket IPC primitives. However, data blocks are passed through shared memory regions to minimize copying.
7 Performance Evaluation

In this section, we report on the performance of the DSM implementation on Unix and Ra. All measurements are done on Sun-3/60 workstations with 4M bytes of memory, connected through a 10M bits/sec ethernet. We mask out the cost of secondary storage access by caching segments in memory before measuring the costs of the DSMC primitives.

7.1 Unix

The implementation on Unix is complete and Table 2 summarizes the results. The Table shows that on an average fetching a segment (without forwarding) of size 8K bytes (the page size on the Sun-3) takes 43.4 ms. Van Renesse et al. report a transfer rate of 40 ms for 8K bytes between two user processes on different nodes using Sun RPC on a 10M bits/sec ethernet [28]. Our implementation uses two user processes per node and still compares favorably with the figures reported by van Renesse et al. A null message from one DSMC to another costs roughly 20 ms, a large portion of which is spent context switching between the kernel and TAL/RaTP, and between TAL/RaTP and DSMC. Moving TAL/RaTP into the Unix kernel would eliminate the additional context switching, and we are currently investigating such an implementation. A semaphore V operation costs only 16.5 ms since it is non-blocking, i.e., the issuing process continues without waiting for the final acknowledgment from the remote DSMC.

As mentioned in §6, we experimented with using socket IPC primitives and System V semaphore primitives to synchronize the TAL/RaTP process and its clients. The numbers reported in this section are from the implementation that used the socket IPC primitives. When using System V semaphores, the average cost of an 8K bytes get request is almost 20 ms more than the cost reported in Table 2. We believe the difference is due to the System V implementation of the semaphore primitives, since the two implementations differ only in the code that synchronizes the TAL/RaTP process and its clients.
### Table 2: Measurements of DSMC operations on Unix

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ms)</th>
<th>Throughput (Kbytes/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V operation</td>
<td>16.5</td>
<td>126</td>
</tr>
<tr>
<td>P operation</td>
<td>31.5</td>
<td>185</td>
</tr>
<tr>
<td>Activate segment</td>
<td>23.3</td>
<td></td>
</tr>
</tbody>
</table>

### Table 3: Ra to Unix Communication Using DSM

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page-in (8K)</td>
<td>35.2</td>
</tr>
<tr>
<td>Segment activation/deactivation</td>
<td>25.1</td>
</tr>
</tbody>
</table>

#### 7.2 Ra

The DSM partition on Ra for handling local requests is complete. This partition communicates with the DSM implementation on Unix to obtain the remote segments. At this point DSMC is not fully ported to Ra to entertain remote segment requests. Table 3 summarizes the preliminary measurements of a Ra node communicating with a Unix node using DSM. Once again the dominant cost in both segment activation/deactivation and page-in is the context switch time at the Unix end. Currently, a null round-trip message time from one Ra node to another through the ethernet is 3.2 ms. These measurements are from an unoptimized implementation, and therefore there is scope for bringing down the message cost. We will have measurements of Ra nodes communicating with one another using DSM shortly.

#### 8 Conclusions and Future Work

We presented an architecture of a distributed shared memory system and described an implementation of the system in the context of the Ra kernel. We also described and reported on the performance of an implementation of the system on Unix. Detailed algorithms for the DSMC primitives, and simulation studies comparing these primitives to RPC are presented in Reference [20]. The utility of these primitives in programming distributed algorithms is illustrated in Reference [23]. So far, our work has concentrated on the mechanisms of distributed shared memory. As part of our future work, we intend to gain more experience with the system and address policy issues such as when to use the RPC mechanism and when to use the DSMC primitives, where to place the semaphore segments, and how to recover from failures.

#### References


A Design of a Memory Management Unit for Object-based Systems*

Umakishore Ramachandran  M. Yousef Amin Khalidi

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Ga 30332-0280 USA

e-mail: rama@gatech.edu yak@gatech.edu

Abstract

Object-based operating systems have the desirable property of separating policies from mechanisms, while providing a protected procedure call interface for accessing system services. However, the kernel mechanisms in such systems rely very heavily on efficient memory management. A set of criteria for supporting the kernel mechanisms in object-based systems is presented. We then present the design of a simple MMU tailored for object-based systems. The proposed design is an engineering solution combining the features available in commercial MMUs.

*This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.
1 Introduction

Traditional systems such as Unix, provide a protected procedure call interface for system services. However, in such systems the kernel tends to be monolithic and bulky since the policies and the mechanisms are all embedded in the kernel. In recent times, especially with the advent of local area networks, message-based operating systems have become popular [CZS3, ABB+86, RR81, HMSC87]. Policies are encoded in server processes and the kernel is lean and supports simply a message passing mechanism between processes. System services are requested by sending and receiving messages. However, the major drawback in such systems is the fact that a round-trip message is involved for requesting system services which can be an order of magnitude more expensive than a protected procedure call [Ram86].

Object-based systems [BHK+88, ABLN85, LS82, Nor87] view the resources of the system as a collection of objects. An object is the encapsulation of the code and data needed to implement the entry points in the object. These entry points provide the procedural interface for an activity to execute the code in the object. System services are requested via invocations (similar to a protected procedure call) on these entry points. At the same time the kernel is lean since it provides simply the invocation mechanism with the policies encoded in system objects outside the kernel. Thus object-based systems, at least in principle, combine the advantages of traditional and message-based systems.

However, implementation of object-based systems on conventional machine architectures can result in performance that is worse than message-based systems [RK89]. Object-based operating systems are critically dependent on efficient memory management. Specifically, an object specifies a virtual space. Virtual spaces are composed of memory segments. Independent virtual spaces can share memory segments or parts thereof. An object invocation entails entering a different virtual space. In a passive object system, there are no processes associated with an object. On invocation, the virtual space of the invoked object is mapped into the invoking process.

Clouds [BHK+88] is an example of such an object-based system. Ra [BHK+89] is an op-
erating system kernel designed to be the nucleus of Clouds. The abstractions that Ra uses for supporting memory management are segments and virtual spaces. A Ra virtual space is described by a segment called the virtual space descriptor (VSD) that contains a collection of windows. Each window is a data structure that identifies the segment that backs a range of addresses in the virtual space (Figures 1 and 2).

An object virtual space is called an O space while a process virtual space is called a P space. Ra views the machine address space as consisting of three distinct regions that are called K, O, and P hardware spaces for kernel, object, and process, respectively. The kernel is always mapped in the K hardware space. A process P space is mapped (or installed) into the P hardware space and unmapped on context switch. On object call, the O space of the invoked object is installed, and on object return, the O space of the caller object is installed.

Ra advocates a model of programming with a large number of possibly small-sized segments. Current implementation of Ra on the Sun-3 allows around 100 windows.

### 2 Background

Clouds is a distributed object-based operating system being developed at Georgia Tech [BHK+88]. The discussion in this paper is presented in the context of Clouds to make it more precise. However, our work is applicable to object-based systems in general.

Clouds is intended to provide a unified environment over distributed hardware. Location independence for data as well as processing, atomicity of distributed computation, and fault-tolerance are some of the research goals of Clouds. Objects and threads are the basic building blocks of Clouds. Objects are passive entities and specify a distinct and disjoint piece of the global virtual address space that spans the entire network. An object is the encapsulation of the code and data needed to implement the entry points in the object. Thus a Clouds object can be considered syntactically equivalent to an abstract data type in the programming language parlance. Access to entry points in the object are accomplished through a capability mechanism in software.
Threads are the only active entities in the system. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object. A thread enters an object by invoking an entry point in the object. It then executes the code in the entry point, and returns to the caller object. Binding the object invocations to the entry points in the object takes place at execution time. A thread in the course of its computation traverses the virtual address spaces of the objects that it invokes.

Ra [BHK87, BHK+89] is an operating system kernel designed to be the nucleus of the Clouds operating system. It currently runs on the Sun-3 architecture. The Clouds operating system is built on top of Ra using a collection of user and system objects. System objects are trusted software modules that reside in system space and provide the critical services that users expect from the operating system such as process control and object invocation. They are used to extend the kernel and to provide the system interface to user objects [BHK+89].

2.1 Ra Primitives

Ra defines and manages three primitive abstractions: segment, virtual space, and isiba. Segments serve as containers of data and may be viewed as uninterpreted sequences of bytes. The contents of a segment may only be accessed when the segment is attached to a range of virtual addresses. Segments persist until explicitly destroyed. Each segment resides in a partition that is responsible for providing backing store for the segment. A partition is a system object that realizes, maintains, and manipulates segments.

Virtual spaces abstract the notion of an addressing domain. A Ra virtual space is described by a segment called the virtual space descriptor (VSD) that contains a collection of windows. Each window is a data structure that identifies the segment that backs a range of addresses in the virtual space (Figures 1 and 2). Windows also describe the protection characteristics of ranges of the virtual space such as read-only or read-write. A virtual space is composed (or built) by a sequence of attach operations, each of which defines one of the virtual space's windows. The attach operation defines a one-to-one mapping between virtual space addresses and segment locations such that virtual addresses starting at the designated address are associated with
segment locations starting at the offset and continuing for the length specified.

The mapped ranges in a virtual space may not overlap, but ranges in the virtual space may remain unmapped. Thus, a single virtual address may not resolve to two or more segment locations but virtual spaces may have "holes" in them. Virtual spaces may share segments or ranges of locations in segments. Also, a segment may be mapped more than once to a single virtual space; that is, two or more windows in a single virtual space may refer to a single segment. Multiple mapping can happen in two ways: either two or more windows in the space refer to disjoint regions of a single segment, or two or more windows in the space refer to overlapping regions of a single segment. In the second case, distinct virtual addresses in the same space resolve to the same segment location.

Ra isibas are an abstraction of the fundamental notion of computation or activity and can be thought of as light-weight processes. Isibas may be used as daemons within the kernel or they may be associated with a virtual space (called P space) to implement a user process. User processes are used to implement Clouds threads.

A Clouds object is a virtual space (called O space) that consists of code and data segments. The code segment of an O space has entry points that can be invoked by user processes.

The machine virtual address space consists of three distinct regions that are called K, O, and P hardware spaces for kernel, object, and process, respectively. The kernel is always mapped in the K hardware space. A Process is mapped (or installed) into the P hardware space and unmapped on context switch.

A process consists of an isiba and a virtual space (P space). The object in which a process is currently executing is mapped into the hardware O space. Object invocation involves mapping the required memory segments of the object into the address space of the invoking process by installing the object as the current O space with the process's P space. On object call, the O space of the called object is installed, and on object return, the O space of the caller object is installed.

Ra advocates a model of programming with a large number of possibly small-sized segments.
Current implementation of Ra on the Sun-3 allows around 100 windows.

3 Comparison Criteria

As can be seen from the description of Clouds and Ra, an efficient implementation relies very heavily on effective management of virtual memory. In particular, the following five MMU features are required to support an efficient implementation of Clouds/Ra:

C1. The ability to cache TLB information across object invocation. The cost of flushing TLB entries at object call/return is an implicit cost that affects mostly user mode time. Systems with one large virtual address space do not require flushing the TLB across object invocations. Examples of such systems include the IBM RT/PC, HP Precision Architecture, and SPUR machine [WEG87].

C2. The ability to implement an object call/return by performing a small number of operations on the MMU. The cost of required MMU operations is an explicit part of object invocation. This cost is quantified by the number of MMU registers and page table entries that need to be modified on each object call/return. In our proposed MMU (see §4), changing one register is the only operation that is required to effect an object call/return. A paper design of object invocation implementation on segmented single virtual address space systems, such as IBM RT/PC and HP Precision Architecture, reveals that only a small number of MMU operations is required [RK88]. However, our experience in implementing Ra on the Sun-3 MMU [BHK+89] reveals that many MMU operations are needed on this machine.

C3. The ability to represent sparse address spaces efficiently. Inefficient representation of sparse address spaces results in page tables that have large memory requirements and lengthy initialization time, relative to the size of address space actually allocated. As mentioned before, page tables organized as trees or inverted tables are suitable for representing sparse address spaces, whereas linear page tables are not.

C4. The ability to share memory among address spaces. This feature is required to support sharing of segments among virtual spaces. Sharing segment in our proposed scheme and
other MMUs with page-tables organized as a tree is easy. MMUs with inverted page
tables (e.g. IBM RT/PC) and virtually-addressed data caches (e.g. SPUR) cannot have
two different virtual addresses mapping into the same physical address (i.e. no aliasing).
Sharing of memory between different spaces is possible only through hardware segment
sharing, which imposes limitations on the number of segments per object and the way they
can be shared between virtual spaces. Clouds advocates a model of programming with a
large number of possibly small-sized segments. Any segment is potentially sharable among
virtual spaces. Because sharing can only be at the hardware segment level, a hardware
segment maps exactly one software segment.

C5. **The ability to allocate/deallocate ranges of addresses easily.** The ability to allocate/deallocate
ranges of addresses is related to sharing of memory segments among spaces, because ranges
of addresses are allocated/deallocated at the segment level. The number of page table and
MMU operations required to allocate/deallocate a range of addresses affects the cost of at-
taching and detaching [BHK+89] segments to virtual spaces. Some systems (e.g. Vax and
Sun-3 MMU) require traversing a potentially large number of page-table entries to allo-
cate or deallocate a range of addresses. Both IBM RT/PC and HP Precision Architecture
adequately support this feature.

These criteria are not satisfied by any of the existing MMUs we studied. A more detailed
qualitative evaluation of commercially available MMUs is presented in [RK88]. Therefore, in
the remainder of this paper we present the design of an MMU tailored for object-based systems.
The proposed design is an engineering solution combining the features available in commercial
MMUs.

4 Proposed MMU

4.1 Design Goals

We used the following goals as a starting point for the design of an MMU tailored to Ra:
• The MMU should meet the five criteria C1–C5 of §3.

• The MMU should be usable for conventional systems as well as object-based systems.

• The MMU should be easy to implement.

• The MMU should be easy to interface to existing CPUs with 32-bit addresses.

4.2 Overview

It is clear that the MMU should cache address translation information across object invocation (criterion C1), and our design achieves this goal by appropriately tagging virtual addresses. There is one important difference between our address tag and other MMUs that tag TLB entries: A distinction is made between O space addresses and P space addresses, and they are tagged differently. In addition, to save on the number of tag bits, addresses are tagged with object and process aliases (see §4.3).

The structure of the page tables affects C2–C5 considerably. MMUs usually dictate a page table format to enable the hardware to search the tables on TLB misses. Unless this page table format meets the OS requirements, a mismatch occurs that may adversely influence performance and increase software complexity. It is important to note that with a large and carefully designed TLB most of the translations come from the TLB, and the page tables are consulted only on TLB misses.

From our Ra implementation experience [BK88], we have found that the VSD structure described in §2 best suites the software implementation. The kernel maintains a VSD structure per (O and P) space. The window descriptors in the VSD describe mappings between address ranges in the virtual space and segments, and an inverted page table is used for each active segment. Though it is possible to implement an MMU that knows about the structure of the VSD, we chose instead a RISC-like approach in the design of our MMU.

Our MMU does not know about page tables nor cares about the structure of the kernel page tables. Instead, it provides a large TLB, with appropriate tagging of virtual addresses, and the
kernel is responsible for reloading the TLB on address translation misses. The structure of the
tables are not known to the MMU nor are they influenced by the MMU design. Instead,
are dictated by the kernel requirements. In case of the Ra kernel, each VS is described by
a VSD, and each segment is described by a hash table. Ra maintains the VSDs, and traverses
them on TLB misses.

4.3 Scheme

Figure 3 depicts the 32-bit CPU virtual address space. The O space resides in the first gigabyte,
while the second gigabyte is reserved for the P space. The upper two gigabytes are reserved for
the K space, and are divided into two sub-spaces, k_v and k_p. Addresses from the k_p sub-space are
deemed as physical addresses and hence do not require translation. The MMU ignores all such
addresses. Addresses from the k_v sub-space are kernel virtual addresses, and their translation
is discussed shortly. Addresses from O and P spaces are tagged with object and process name
aliases, respectively.

Figure 4 shows the structure of the MMU. The Object Name Register (ONR) and Process
Name Register (PNR) hold the names (a bits long) of the current object and process, respectively.
The kernel is responsible for setting these registers on object invocations and process context
switches. The Alias Table (AT) is maintained by the MMU. The AT maintains the names of
the last \(n\) used object and process names. The index of a space name into the AT is used as a
name alias for addresses coming from this space. The Object Alias Register (OAR) and Process
Alias Register (PAR) contain the aliases (\(1 \log n\) bits long) for the values stored in ONR and PNR,
respectively. The AT enables the MMU to store a smaller tag with each address instead of a
full object/process name.

The structure of a TLB entry is shown in Figure 4. Each entry consists of a tag of size \(\log a\)
bits, a virtual page number (VPN) of size \(v\) bits, a physical page number (PPN) of size \(p\) bits.
and eight 1-bit flags. When the global flag (G) is set, the VPN is global (kernel virtual address)
and will match any valid TLB entry that has the same VPN, regardless of the tag value. When
the dirty flag (D) is set, it indicates that the page has been written into. The MMU generates
a fault when a write is attempted and D is not set. This fault is necessary to enable updating of the corresponding page table entry in the software data structures maintained by the kernel. When the valid flag (V) is set it indicates that this TLB entry is valid. The two protection bits (PR) encode one of four possible page protection modes, as shown in Table 1. The history bit (H) is used by the MMU to implement a pseudo least-recently used algorithm [Mot86] that decides which TLB entry to replace when \texttt{WriteTLBentryLRU} command is issued. The two software flags (S) can be set by the software and have no meaning for the MMU.

Each address generated by the CPU is viewed by the MMU as shown in Figure 5. The high-order two bits are used to select one of the four spaces: O, P, k_v, and k_p. Address translation proceeds as shown in Table 2: Addresses from O space (high order two bits=00) are tagged with the contents of OAR. P space addresses (high order two bits=01) are tagged with the contents of PAR. Addresses from k_v sub-space (high order bits=10) are tagged with 0. Addresses from k_p sub-space (high order bits=11) are ignored by the MMU. A tagged address matches a valid TLB entry if (a) the tag and VPN of the generated address matches the tag and VPN of the TLB entry, or (b) the VPN of the generated address matches the VPN of the TLB entry and the G bit is set.

An implementation that uses our MMU ties a portion of the k_p sub-space directly to physical memory. The k_p sub-space is intended for use by those portions of the kernel that need not be mapped dynamically. Specifically, most of the kernel consists of code, constants, and static data that need not be remapped once loaded into memory. By eliminating the need to translate addresses from those portions of the kernel, the poor locality properties of system kernels observed by Clark and Emer [CE85] do not affect the TLB. A similar approach is taken by the MIPS MMU [DMM86]. The disadvantage of this approach is a loss of some degree of protection, because kernel software can inadvertently overwrite read-only portions of the kernel. However, kernel software is expected to be correct, and we do not expect this to be a major disadvantage. In §4.4 we show how to utilize this feature in the Ra kernel.

The kernel modifies the ONR to map a new O space on object call/return and process switches, and modifies the PNR to map a new P space on process switches. When the ONR
(PNR) is changed by the software, the MMU searches the AT for an entry with the same value as the ONR (PNR), and OAR (PAR) is set to this value. If no matching entry is found in the AT, an entry is reused, and all TLB entries with the same tag as the index of the chosen entry are flushed (entries with the G bit set are not flushed). The MMU flushes a TLB entry by clearing the V bit.

The software and MMU communicate through the memory mapped registers shown in Figure 6, using the command set summarized in Table 3. The kernel writes the arguments of a command into the appropriate registers, writes the command into CommandReg, then reads back output arguments (if any) from the appropriate registers. The ReadTLBentry and WriteTLBentry commands are used by the kernel to read/write TLB entries. WriteTLBentryLRU is used to overwrite the the first invalid entry or the least-recently used entry. ProbeTLB is used by the kernel to search the TLB for an entry, given a virtual address. Note that if the G bit is set in the TLBentryReg, ProbeTLB ignores the value of the tag, and returns the index of any valid entry that matches the given VPN. ProbAT returns the tag associated with the given process/object name. FlushTLB flushes all TLB entries with the given tag (and G=0). SetONR and SetPNR write a new value into ONR and PNR, respectively.

The MMU interrupts the CPU (via an interrupt line) in one of three cases: When an address does not match any entry in the TLB, a miss-fault is generated; when a protection violation is detected, a protection-fault is generated; and when a write is attempted to a page with the D bit cleared, a dirty-fault is generated. It is up to the kernel to distinguish the type of the fault by examining TLBentryReg and VirtAddrReg. We chose this simple mechanism to allow our MMU to be easily interfaced to a variety of microprocessors. On a miss-fault, the virtual address causing the fault is written into the VirtAddrReg. On a protection-fault or a dirty-fault, the TLB entry of the requested page is written into TLBentryReg, and a value of 0xffffffff is written into the VirtAddrReg. Therefore, if a value other than 0xffffffff is read from the VirtAddrReg, the kernel discerns the fault to be a miss-fault. Otherwise, the D bit in the TLBentryReg determines whether the fault is a dirty-fault or a protection-fault. Note that the address 0xffffffff is in k_p and can never generate a miss-fault because the MMU ignores all
An implementation of the MMU has to fix the various parameters of the chip, such as the page size and number of TLB entries. We expect the MMU parameters to range over the following values: From our experience in building the Clouds kernel on the Vax architecture (page size=512 bytes) and the Ra kernel on the Sun-3 architecture (page size=8K bytes), we believe that a page size in the range 2K-4K bytes is an appropriate size. From a simulation study of our MMU design [Kha89], an AT size in the range 16-32 entries, and a TLB size in the range of 128-256 entries are appropriate. Ten bits should be an appropriate size for object/process names, given that the kernel can use the index of the object/process descriptor as its name (see §4.4). Table 4 lists these values.

4.4 Implementation of Ra using proposed MMU

The proposed MMU is well-suited to the Ra kernel. The Ra software machine-independent structures that describe each virtual space (VSD) and segment are sufficient for representing virtual space mappings, and no hardware-dependent structures are needed. The kernel is required to access the MMU in the following cases:

- **Object invocation and process switches.** On object invocation, the ONR is set to the index of the VSD of the object to be mapped. Each VSD in Ra has an index [BK88], and this index can be used for the value for ONR. Similarly, on process context switches, PNR is set to the index of the VSD describing the P space to be mapped.

- **MMU faults.** On a miss-fault, the kernel searches the VSD (as described in [BK88]) to locate the mapping information for the faulting address. As part of the normal page-fault handling, the kernel checks the request for validity, and may have to access a partition to bring the required page into memory. When the mapping information is available, it is written into the TLB using **WriteTLBentryLRU** command. On a protection-fault, the kernel has the VPN available in TLBentryReg, and can decide on the proper action. On a dirty-fault, the kernel should set the D bit using **WriteTLBentry**, after setting the
corresponding bit in the software physical-page descriptor [BK88]. Note that copy-on-write semantics can be implemented upon detection of dirty-faults.

- **Reuse of a VSD.** When the kernel reuses a VSD, it must issue a FlushTLB command to make sure any entries from the destroyed VSD are purged.

- **Modifying the mapping of a page.** When the the mapping of a page is modified (e.g. on a page-out operation), the corresponding entries in the TLB mapping this page has to be flushed. The commands ProbeTLB, WriteTLB, and ReadTLB are provided for this purpose.

To exploit the \( k_v \) and \( k_p \) portions of the K space no modifications to the architecture of Ra is required. Only the system-object loader [BHK+89] that is responsible for loading system objects in the K space needs to be modified. The loader should be aware of the differences between \( k_v \) and \( k_p \), and any portion of the system that may be remapped dynamically has to reside in \( k_v \).

### 4.5 Evaluation

The approach to handle TLB reloading in software is not novel. Others have advocated a similar software approach to handle address translation misses [DMM86, MLM+86, CGBGSS]. Among the advantages cited for this approach is the simplification of circuitry, and saving in chip area [DMM86]. The saving in chip area offers more space for a larger TLB.

This approach enables the MMU to match the OS requirements by using the best suited page table format as dictated by the software. By narrowing the semantic gap between the MMU and the operating system kernel, the programming model of the system is not hindered by the underlying hardware. In addition, the virtual memory system in the kernel is considerably simplified.

The cost of handling TLB misses in software is not a dominating cost as may first seem. The cost of TLB refill in software on the MIPS CPU is around 1.7 \( \mu \)seconds [DMMS6] (using an 8 MHz CPU and a simple 2-level page table). In comparison, the cost of a TLB refill in hardware on the Vax-11/780 is around 4.4 \( \mu \)seconds. Cheriton et al. [CGBGSS] estimated the
cost of servicing an address translation miss on the VMP multiprocessor (using a software page table structure similar to ours) to be 10 $\mu$seconds on an average, versus about 4 $\mu$seconds with a conventional page table. By their estimates, this scheme adds about 4.8 nanoseconds on an average to each memory reference, which is an acceptable overhead given that each memory access costs 75 nanoseconds on VMP [CGBG88].

Note that when a TLB miss-fault is a result of a missing page (i.e. the required page is not in main memory), the cost of bringing the missing page into memory overshadows the extra overhead of the software-controlled TLB (bringing a page into memory costs on the order of tens of milliseconds). Appropriately tagging the TLB entries, a large page size, and the use of $k_v$ and $k_p$ sub-spaces, help keep the TLB miss rate low (similar arguments are made in [DMMS86]).

We evaluate the proposed MMU with respect to the criteria of §3:

C1. The MMU caches address translation information across object invocation and process context switches. By separately tagging O and P space addresses, translation information is kept in the TLB when O and/or P spaces are changed. We evaluated our scheme using trace-driven simulation, and compared its performance to the usual case of flushing the TLB on each object call and return [Kha89]. The simulation study used synthetic address traces and measured the effect of caching O space addresses only. The results of our performance study is summarized below, where $T_1$ refers to our scheme, while $T_0$ refers to the usual case of flushing the TLB on object invocations:

- For a given TLB size, the miss ratio of $T_1$ is consistently less than the miss ratio $T_0$. For a TLB size of 128, the miss ratio of $T_1$ is 100 times less than the miss ratio of $T_0$.

- The $T_1$ miss ratio decreases as the TLB size increases beyond the point when the $T_0$ scheme miss ratio levels off. $T_1$ uses the additional TLB entries to cache more address translations that may be used in the future, unlike $T_0$ which flushes cached translations on each object call and return.

- Depending on the ratio of $T_h/T_m$ (where $T_h$ is the hit cycle time, and $T_m$ is the miss
cycle time), the overall percentage improvement of T1 over T0 ranges from around 15% to 30%.

C2. One operation is required to map an O space, and at most two operations are needed for a process context switch.

C3. The MMU does not impose a page table structure, and the kernel is able to use an efficient representation of virtual spaces.

C4. The MMU allows aliasing and no restrictions are imposed on sharing memory among spaces.

C5. Allocating a range of addresses does not require MMU intervention (though the software may preload TLB entries). Deallocating a range of addresses requires the kernel to flush all entries in the TLB in the deallocated range. Because there is no practical restriction on the number of segments attached to a virtual space, we expect the number of attach/detach operations to be small. We expect that attaching small segments for passing parameters and temporary manipulation of segments to be more frequent than attaching/detaching large segments. Using the ProbeTLB and WriteTLBentry, the kernel can detach a segment in a small number of operations (proportional to the number of pages in the window mapping the segment).

5 Conclusions

Shared memory is a simple and powerful paradigm for structuring systems. Object-based operating systems embody this paradigm as the basic building block in providing a clean separation between policies and mechanisms. The discussion in this paper reveals some of the inadequacies of commercial MMUs for supporting the requirements of object-based operating systems. The design presented in this paper shows that it is possible to meet these requirements with a combination of the features available in commercial MMUs.
References


Virtual Space defined by descriptor

Figure 1: Ra Virtual Space

<table>
<thead>
<tr>
<th>PR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Kernel read-only access; user no access</td>
</tr>
<tr>
<td>01</td>
<td>Kernel read-write access; user no access</td>
</tr>
<tr>
<td>10</td>
<td>Kernel read-write access; user read-only access</td>
</tr>
<tr>
<td>11</td>
<td>Kernel read-write access; user read-write access</td>
</tr>
</tbody>
</table>

Table 1: Protection Bits

<table>
<thead>
<tr>
<th>Value of bits 31 &amp; 30</th>
<th>MMU Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Tag VPN with OAR and compare</td>
</tr>
<tr>
<td>01</td>
<td>Tag VPN with PAR and compare</td>
</tr>
<tr>
<td>10</td>
<td>Tag VPN with 0 and compare</td>
</tr>
<tr>
<td>11</td>
<td>Ignore address</td>
</tr>
</tbody>
</table>

Table 2: Address translation process

17
<table>
<thead>
<tr>
<th>Command</th>
<th>Input</th>
<th>Output</th>
<th>Side Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadTLBentry</td>
<td>IndexReg</td>
<td>entryReg</td>
<td></td>
</tr>
<tr>
<td>WriteTLBentry</td>
<td>IndexReg, entryReg</td>
<td></td>
<td>TLB entry updated</td>
</tr>
<tr>
<td>WriteTLBentryLRU</td>
<td>entryReg</td>
<td></td>
<td>LRU TLB entry updated</td>
</tr>
<tr>
<td>ProbeTLB</td>
<td>entryReg</td>
<td></td>
<td>If found, output in index field of IndexReg</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Otherwise, P is set</td>
</tr>
<tr>
<td>ProbeAT</td>
<td>NameReg</td>
<td></td>
<td>If found, output in tag field of t hi</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Otherwise, P is set</td>
</tr>
<tr>
<td>FlushTLB</td>
<td>tag field in entryReg</td>
<td></td>
<td>All entries with same tag and G=0 are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>invalidated</td>
</tr>
<tr>
<td>SetONR</td>
<td>NameReg</td>
<td></td>
<td>ONR updated</td>
</tr>
<tr>
<td>SetPNR</td>
<td>NameReg</td>
<td></td>
<td>PNR updated</td>
</tr>
</tbody>
</table>

Table 3: Proposed MMU commands

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>128 – 256 entries</td>
</tr>
<tr>
<td>n</td>
<td>16 – 32 entries</td>
</tr>
<tr>
<td>a</td>
<td>10 bits</td>
</tr>
<tr>
<td>Page size</td>
<td>2K – 4K bytes</td>
</tr>
<tr>
<td>v</td>
<td>19 – 18 bits</td>
</tr>
<tr>
<td>p</td>
<td>18 bits</td>
</tr>
<tr>
<td></td>
<td>(0.5G – 1G bytes of physical memory)</td>
</tr>
</tbody>
</table>

Table 4: Proposed MMU parameters
Figure 2: Structure of a window

\[
\begin{array}{|c|c|}
\hline
\text{Window } i & \\
\text{Starting addr in VS} & \\
\text{Ending addr in VS} & \\
\text{Segment} & \\
\text{Starting offset in Segment} & \\
\text{Protection Information} & \\
\hline
\end{array}
\]

\[2^{32} - 1\]

\[k_p\]

\[1\text{G bytes}\]

\[\text{not translated through TLB}\]

\[k_v\]

\[1\text{G bytes}\]

\[G\text{ bit set}\]

\[P\text{ space}\]

\[1\text{G bytes}\]

\[\text{tagged with process alias}\]

\[O\text{ space}\]

\[1\text{G bytes}\]

\[\text{tagged with object alias}\]

Figure 3: CPU virtual address space of proposed MMU

Figure 4: Proposed MMU

Figure 5: CPU Virtual Address
Figure 6: Proposed MMU communication registers
Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer

Umakishore Ramachandran  Mustaque Ahamad  M. Yousef A. Khalidi

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Ga 30332-0280 USA

Abstract

Clouds is a distributed operating systems research project at Georgia Tech. With threads and passive objects as the primary building blocks, Clouds provides a location-transparent protected procedure-call interface to system services. Mechanisms for synchronization within objects, and atomicity of computation are supported in Clouds. The primary kernel mechanism in object-based systems such as Clouds is the mapping of the object into the address space of the invoking thread. Hence the performance of such systems depends crucially on the efficiency of memory mapping. The problem gets exacerbated with distribution since now the object invoked by a thread may be located on a remote node. Since a thread can potentially invoke any object, the virtual address spaces of all objects can be viewed as constituting a "global distributed shared memory". Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-mapping mechanism. We present an organization and mechanisms for supporting this abstraction of a distributed shared memory. We propose a distributed shared memory controller that provides mechanisms for efficient access and consistency maintenance of the distributed shared memory. The novel feature of our approach is the exploitation of process synchronization to simplify consistency maintenance. The distributed shared memory mechanisms serve as the backbone for implementing object invocation, synchronization mechanisms, and network-wide memory management in the Clouds system.

1 Introduction

We are exploring hardware support to improve the performance of object-based distributed operating systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network. There are one or more processors and a certain amount of memory in each node. Nodes do not share memory; message exchange across the network is the only mechanism for communication between them.

Many operating systems designs for such an environment [4,14,20,32] place a message-passing kernel on each node, supporting processes and communication between them via explicit messages. This kernel supports both local communication—communication between processes on the same node—and nonlocal or remote communication (sometimes implemented via a distinguished network manager process). Access to system services are requested via protected procedure calls in a traditional system, whereas in a message-based operating system they are requested via message passing. While a simple procedure call costs just a few instructions, and a protected procedure call (kernel call) costs a few hundred instructions, IPC costs a few thousand instructions in several systems that we studied [28]. Message-based operating systems are attractive for structuring distributed systems due to the separation of policy (encoded in server processes) from mechanism (in the kernel).

Object-based distributed operating systems [7,2,39,34,26] view the resources of the system as a collection of objects. Clouds [7] is an object-based distributed operating system being developed at Georgia Tech. In Clouds, system services are encoded in passive objects (syntactic units that are similar in flavor to the server processes of message-based systems) that occupy distinct virtual address spaces in the system. Access to system services are requested by invocations (similar to the protected procedure calls of traditional systems) into the appropriate system object. The speed of object invocation is often used as a yardstick for measuring the performance of object-based systems. In passive object-based systems, invocation performance depends on the efficiency of object memory management (see §2). The problem gets exacerbated with distribution since now the invoked object may be located on a remote node.

In this paper we suggest mechanisms (that can be implemented in hardware/firmware) for supporting the abstraction of "globally distributed shared memory". In §2, we give the relevance of our work and motivate the need for supporting this abstraction. Related work in the area of memory coherence is presented in §3. With the Clouds operating system as our target application, we present our ideas on customizing the memory coherence requirements in §4. Our proposed hardware organization, the primitives provided by the distributed shared memory controller, and the algorithms for maintaining the consistency of the distributed shared memory are discussed §5. In §6 we describe a software implementation of the proposed primitives. A performance evaluation of our scheme is presented in §7. Finally, our conclusions are presented in §8.

2 Relevance

Clouds [7] is a distributed operating system that is intended to provide a unified environment over distributed hardware. Location independence for data as well as processing, atomicity of distributed computation, and fault-tolerance are some of the research goals of Clouds.

Objects and threads are the basic building blocks of Clouds. Objects are passive entities and specify a distinct and disjoint piece of the global virtual address space that spans the entire network. An object is the encapsulation of the code and data needed to implement the entry points in the object. Thus a Cloud's object can be considered syntactically equivalent to an abstract data type in the programming language parlance. Access to entry points in the object is accomplished through a capability mechanism in the kernel.

Threads are the only active entities in the system. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object. A thread enters an ob-
ect. A thread enters an object by invoking an entry point in the object. It then executes the code in the entry point, and returns to the caller object. Binding the object invocations to the entry points in the object takes place at execution time. A thread in the course of its computation traverses the virtual address spaces of the objects that it invokes. A thread is comparable to a process (as defined in many conventional systems), with the exception that a thread may span machine boundaries. For the purposes of this paper, a process is synonymous to a thread.

The virtual address spaces of all objects can be viewed as constituting a "global distributed shared memory". Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-mapping mechanism. Local object invocation involves mapping the required memory segments of the object into the address space of the invoking thread. The current trend in structuring distributed systems is to use a collection of diskless computational servers (workstations) and a few data servers (file servers). In such an environment, the code and data for the (local) invocation has to bepaged-in from the data server. Further, for remote object invocation we have one of two choices: The first choice is to perform the computation at the node where the object resides (remote procedure call). The second choice is to make the invocation appear local by bringing in the segments required for the invocation. While we have to support the former for immovable objects (such as an object that reads disk blocks), we believe that the latter may be a better choice for movable objects. There are two reasons to support this belief:

- the principle of locality [10] that suggests an invocation (or other invocations in the same object) may be repeated
- the reduction in computational overhead due to the elimination of slave process management to support remote invocation at the node where the object resides [23].

The idea of using the abstraction of a globally distributed shared memory in an object-based system appears to be novel. Several other researchers have proposed software architectures based on the shared memory paradigm, in different settings:

- In a speech recognition application, Bisiani and Forin [9] use data structures that are shared by multiple language modules that are distributed on heterogeneous machines. They show that communication through shared memory is a viable alternative to message-passing even when the environment involves cooperation between multilingual program modules and heterogeneous machines.

- Processes in the programming language Linda [10,18] communicate via a globally-shared collection of ordered tuples.

- A (logically shared) bulletin-board is proposed by Birman, et al. [8] for structuring asynchronous interactions between processes in distributed systems.

- To exploit the multicast capability of local area networks, Ahamad and Bernstein [1] describe a bulletin-board based interprocess communication mechanism that could just as well be implemented with a shared memory paradigm.

- By integrating the mechanisms for virtual memory management and local interprocess communication, Mach [32] achieves efficient implementation of local interprocess communication. Currently, researchers at CMU are investigating the use of shared memory mechanisms (such as Linda) for inter process communication in the context of network communication as well [38].

- Zayas [39] achieves substantial reduction in the cost of process migration by using copy-on-write techniques (Accent [31]) and on-demand fetches during remote execution.


- Emerald [21] is a distributed object-based language and system with support for object mobility.

We believe that software architectures based on the shared memory paradigm would benefit considerably (in both performance and ease of implementation) if the underlying hardware were to provide a transparent mechanism for efficient access and consistency maintenance of distributed shared memory. Thus the investigation of hardware support for providing the abstraction of a distributed shared memory is worthwhile.

3 Related Work

3.1 Shared Memory Multiprocessors

Consistency maintenance in distributed shared memory is similar to cache coherence in multiprocessors. Shared memory multiprocessors such as Encore's Multimax [16], consist of several processors connected to a common shared memory via a system bus. A main memory cache is associated with each processor to help reduce the traffic to the shared memory. Multiprocessor cache consistency protocols (such as [19,22]) ensure the following coherence constraint: a read operation performed by a processor returns the most recent value written into that location (by any processor). This criterion is appropriate in a shared memory multiprocessor since the system bus (a broadcast medium) serializes the memory operations of all the processors. This approach is a brute-force one to assure coherence since there is no semantic knowledge associated with the data being cached.

We make the following observations regarding these protocols: Multiprocessor cache coherence algorithms (see [3] for a survey) consider memory coherence problem in isolation. In reality, memory coherence and process synchronization are closely intertwined. The ability of a process to read or write shared data is invariably acquired through some synchronization method. Since the cached data has no synchronization information associated with it, these algorithms tend to be an overkill owing to their generality. However, these algorithms are a viable approach for solving the cache coherence problem in multiprocessors since the cost (measured in circuit complexity as well as time) of implementing them in hardware is a small fraction of the total system cost. Further, multiprocessors have the ability to invalidate all cached copies in one atomic bus cycle owing to the system bus.

3.2 Distributed Shared Memory

Some work has been done in extending the shared memory paradigm to a distributed system. Li [24] presents a slight variation of the Berkeley protocol for multiprocessor cache consistency [22] as a solution to maintain the consistency of distributed shared memory. The entire memory is considered potentially sharable for both reads and writes. Hence the current owner of a page (the node that has write access to the page) keeps a copy-set—the set of nodes that have a read-copy of the page. In Li's scheme, a write into a shared location results in an invalidation message to be sent to each of the nodes in the copy-set for the page containing the location. Note that the nodes that have a copy may never be interested in that page. These invalidation messages are a high price to pay in a distributed environment. Some of the pages may never be written into (for example, text pages), but since the algorithm only deals with raw pages the overhead of
keeping the copy-set information is also incurred for such pages. Li's solution has the same drawback as the multiprocessor cache coherence algorithms—memory coherence problem is dealt with in isolation without considering process synchronization.

Agora [9] (see §2) supports shared data structures that span heterogeneous machine architectures and multiple languages. Agora adopts a mechanism similar to Li's with invalidation messages on writes to shared locations, and hence is inefficient for the same reason. However, since Agora is tailored for a very specific application it allows sharing at the level of individual data structures rather than raw pages.

Fleisch [17] proposes a distributed shared memory facility for the Locus [26] system that supports Unix System V shared memory semantics. The proposed coherence algorithm is similar to Li's [24]. Fleisch's work does not address such issues as locality and synchronization.

4 Customizing Coherence

We are interested in exploring hardware support for the abstraction of a distributed shared memory. The basic idea is the following: Each node has associated with it a “network cache”—a repository for recently accessed remote memory-segments and their owners (nodes). If a remote memory-segment is not in the cache, it is requested from the owner and cached for future reference. If the segment is "dirtied", the network-cache becomes the supplier in the future for this segment. If and when the segment is replaced it is sent back to the owner of the segment.

How do we define memory coherence in this environment? The definition that works well for a shared memory multiprocessor is inappropriate for this environment since there is no “system bus” to impose a total order on the memory operations that are performed by all the processors. Further, while invalidation of cached copies of data is a viable approach in multiprocessors (with a system bus) it is infeasible (due to the cost of the invalidation messages) in a distributed system. Invalidation involves at least sending a multicast message to all the nodes that have a read-copy of the segment. Achieving reliable delivery of such multicast messages is prohibitively expensive in a distributed system [13].

Exploiting application specific semantic information would reduce the complexity of the problem and make a hardware solution viable. In the scheme that we propose we deal with process synchronization and memory coherence together. Since the application that we are trying to support is the Clouds operating system we cannot make any more assumptions as Agora does in providing finer-grain sharing at the level of individual data structures. Read-only code and data areas can be distributed without the need for maintaining consistency. Only the read-write data area of an object requires maintenance of consistency.

5 Hardware Organization

The organization (Figure 1) we propose inside each node of the network is the following: a host that executes distributed applications; a distributed shared memory controller (DSMC) together with the network interface assists the host in mapping memory segments (local and remote) into the virtual address spaces of the application processes. There is a minimal kernel on the host that traps system calls, and virtual address translation faults. The DSMC is also in control of the network. The system memory is (logically) partitioned into two parts: One part (object memory) is for housing the segments of locally created objects; the other part (network cache) is for caching segments from remote objects. Conceptually there are two lists of process control blocks: the host-list that the host looks at to schedule runnable processes; the DSMC-list that the DSMC looks at to service memory segment requests (local and remote). The DSMC enqueues processes that have become runnable again (after the fault service) in the host-list. The operations provided by the DSMC, and the algorithms for implementing these operations are the topics of interest in this paper. Although conceptually the DSMC is shown as a "co-processor", it can be implemented as a software module that coexists with the kernel (see §6).

5.1 Clouds Objects

Objects in Clouds consist of one or more of the following types of areas: read-only code, read-only data, shared read-write data. We refer to these areas as segments. Segments serve as containers of data and can be of variable size. The contents of a segment may only be accessed when the segment is attached to an object [6]. Segments persist until explicitly destroyed. We refer to the node where a segment is created as the owner of the segment. There is a well-known area of the system memory—keeper segment—maintained by the DSMC. We associate a keeper location with each read-write segment owned by the node. The keeper location for a segment points to the node currently having write access to the segment. Owner and keeper point to the same node at segment creation time.

5.2 Virtual Address

The virtual address generated by a process is interpreted as being composed of three fields: object name, segment name, and segment offset. Using the segment name, the DSMC does a table lookup to determine the location of the segment. The DSMC at each node keeps a segment table. This table contains the following information for segments currently mapped at the node: their sizes and types, and their mapping to physical segments or disk blocks. Further, for locally created segments the segment
table remembers the current location (keeper) of the segment. The segment-table entries also each have a queue—the list of processes waiting on the segment.

A segment fault occurs when the virtual address generated by a process is currently unmapped. In addition, every object invocation results in a fault to map in the segments needed for the invocation. A segment fault (or object invocation) manifests as a trap into the kernel on the host. As a result of the trap, the kernel calls an object manager that in turn invokes the DSMC primitives. The host enqueues the faulting process in the DSMC-list (for the DSMC), and schedules the next runnable process. The DSMC is responsible for mapping in the required segments and making the process runnable. A process waiting for a remote segment is queued on the appropriate entry in the map table. On receiving a remote segment in response to a previous request, the DSMC sets up the memory map of the processes waiting for that segment and dequeues these processes in the host-list (for the host).

Though the segment as used in this paper is a logical entity, it is mapped onto the underlying hardware into an integral number of pages. In Reference [6], we discuss an implementation of the DSMC as a software module that is part of the paging system. This implementation is summarized in §6.

5.3 DSMC Primitives

The DSMC provides data transfer and synchronization primitives for supporting the abstraction of a global distributed shared memory. From the applications (Clouds kernel in our case) DSMC entertains four types of requests:

1. get(segment): The DSMC is responsible for fetching the required segment, setting up the segment table of the faulting process, and enqueuing the process on the host-list for execution.

2. discard(segment): The DSMC frees the physical memory occupied by the segment by sending the segment back to its owner.

3. P(segment, semaphore): The DSMC performs an atomic semaphore P operation on the specified synchronization variable.

4. V(segment, semaphore): The DSMC performs an atomic semaphore V operation on the specified synchronization variable.

The DSMCs exchange messages to satisfy these requests. The messages recognized by the DSMC are the following:

msg_get(segment, mode)
msg_segment(segment)
msg_forward(segment)
msg_discard(segment)
msg_P(segment, semaphore index i)
msg_V(segment, semaphore index i)
msg_ack(segment, semaphore index i)
msg_error(segment, error type)

In the next few subsections, we give the algorithms required for implementing the data transfer and synchronization primitives. For ease of understanding the algorithms, we show a single thread of execution while processing these primitives. However, it should be noted that the DSMC is multiprogrammed. For example, on a get request the local DSMC sends a msg_get request to its peer and then proceeds to the next request in its queue (DSMC-list). Eventually, when a reply arrives from its peer the local DSMC takes the appropriate action. In the algorithms to follow, we do not show this asynchrony. For simplicity, we show each request from a DSMC to its peer as a synchronous one.

5.3.1 Data Transfer

Get(segment) is the primitive for mapping a currently unmapped segment. On a segment fault the host queues the faulting process in the DSMC-list with a get request for the required segment. A get request for a read-only segment is trivially satisfied if the object is local; if the object is remote the DSMC requests its peer (at the owner node) for the read-only segment. On receiving the request, the peer DSMC sends the read-only segment to the requesting DSMC without performing any house-keeping work.

A get request for a read-write segment is implemented by the following algorithm:

if local(segment) then
   case keeper of
      self:
         if not memory(segment) then
            bring segment into object-memory from disk;
            endif;
         remote:
            send msg_get(segment) to remote DSMC;
            receive msg_segment(segment) from remote DSMC;
            place segment in object-memory;
            change keeper to self;
            endcase;
      else /* segment is remote */
         /* go ask the owner */
         send msg_get(segment) to remote DSMC;
         receive msg_segment(segment) from remote DSMC;
         place segment in network-cache;
         endif;
         /* the requested segment is now in memory; map it into the process' address space */
         map segment into process' address space;
   endif;
else /* segment is remote */
   /* note at most one forwarding */
   change keeper to requesting DSMC;
   send msg_forward(segment) to keeper;
   /* note at most one forwarding */
   change keeper to requesting DSMC;
   endcase;

When a DSMC receives msg_get request for a (read-write) segment it does the following:

case keeper of
   self:
      if not memory(segment) then
         bring segment into object-memory from disk;
         endif;
      change keeper to requesting DSMC;
      send msg_segment(segment) to requesting DSMC;
      remote:
         send msg_forward(segment) to keeper;
         /* note at most one forwarding */
         change keeper to requesting DSMC;
         endcase;

The owner may forward the request to the current keeper. A DSMC that receives a msg_forward request for a (read-write) segment does the following:

send msg_segment(segment) to requesting DSMC;
invalidate the segment entry in the segment-table;
return freed segment in the network-cache to free-list;
Placing a segment in the network-cache may involve freeing up segments from the network-cache. The DSMC sends \texttt{msg\_discard} to the owner DSMC of the segment for this purpose. The algorithm for freeing up a segment from the network-cache is the following:

invalidate the segment entry in the segment-table;
if read-write\texttt{(segment)} then
  if dirty\texttt{(segment)} then
    send \texttt{msg\_discard}\texttt{(segment)} enclosing segment to owner;
    else /* clean segment */
    send \texttt{msg\_discard}\texttt{(segment)} to owner;
endif;
endif;
The owner DSMC upon receiving \texttt{msg\_discard} request does the following:

if segment enclosed then
  if memory\texttt{(segment)} then
    write segment into memory;
    else
      write segment onto disk;
    endif;
endif;
change keeper to self;

Due to network delays it is possible that a node (the future keeper of a segment) may receive a \texttt{msg\_forward} request before the segment arrives from the owner or the forwarder (the current keeper). However, recall that the keeper information with the owner of an object is absolute. Therefore the node can simply buffer the segment-request, and honor it when the segment arrives, without compromising the correct operation of the system.

5.3.2 Synchronization

Get, and \texttt{discard} are not enough to efficiently implement the synchronization primitives provided by the Clouds kernel. For example, consider semaphore operations \((P\texttt{ and } V)\) supported by the Clouds kernel for synchronization inside an object. With just \texttt{get} and \texttt{discard} operations supported by the DSMC, an obvious implementation of the semaphore operations may be the following: place the semaphore data structure (a value field, and a list of waiting processes) in a read-write segment; on every semaphore operation \texttt{get} this data structure from its keeper; and perform the operation atomically (disabling interrupts). By definition, semaphore operations order the execution of cooperating processes. Therefore, it seems wasteful to ship the semaphore data structure back and forth between these processes when they are on different nodes.

For now we consider the solution where the semaphore operations are performed at the owner node. The semaphore operations are provided as primitives understood by the DSMC. Synchronization variables are allocated semaphore segments. We note the fundamental difference between the roles played by the DSMC in the data transfer operations \((\texttt{get}, \texttt{discard})\) and the synchronization operations \((P, V)\): In the former, the DSMC fetches the current copy from the keeper (it does not have to worry about the contents of the fetched segment); whereas in the latter the DSMC manipulates the contents of the synchronization variables at the owner node and reports success/failure to its requesting peer DSMC.

When a process performs an operation \((P\texttt{ or } V)\) on a synchronization variable, it results in a trap to the kernel. The kernel in turn enqueues this process in the DSMC-list. The algorithm executed by the DSMC for implementing \(P\texttt{(segment, semaphore }i)\) or \(V\texttt{(segment, semaphore }i)\) operation is the following:

if local\texttt{(segment)} then
  case operation of
    \(P:\)
      decrement synchronization variable \(i\);
      if variable value less than zero then
        enqueue process on queue\texttt{(segment, }i\texttt{)};
        else
          enqueue process on host-list; /* ready to resume execution */
    endif;
    \(V:\)
      increment synchronization variable \(i\);
      if variable value less than or equal to zero then
        remove an entry from queue\texttt{(segment, }i\texttt{)};
      endif;
      case entry of
        remote DSMC:
          send \texttt{msg\_ack\texttt{(segment, }i\texttt{)}} to remote DSMC;
          process:
            enqueue process on host-list; /* removed entry */
          endcase;
        \(P:\)
          send \texttt{msg\_P\texttt{(segment, }i\texttt{)}} to remote DSMC;
          dequeue process from queue\texttt{(segment, }i\texttt{)};
          enqueue process on host-list; /* ready to resume execution */
        \(V:\)
          send \texttt{msg\_V\texttt{(segment, }i\texttt{)}} to remote DSMC;
          enqueue process on host-list; /* V-ing process */
      endcase;
    else /* remote segment */
      case operation of
        \(P:\)
          send \texttt{msg\_P\texttt{(segment, }i\texttt{)}} to remote DSMC;
          enqueue process on queue\texttt{(segment, }i\texttt{)};
          receive \texttt{msg\_ack\texttt{(segment, }i\texttt{)}} from remote DSMC;
        \(V:\)
          enqueue process on host-list; /* ready to resume execution */
        endcase;
    endif;
endif;

On receiving a \texttt{msg\_P} or a \texttt{msg\_V} request the DSMC does the following:

  case operation of
    \(P:\)
      decrement synchronization variable \(i\);
      if variable value less than zero then
        enqueue (requesting DSMC, process) on queue\texttt{(segment, }i\texttt{)};
      else
        send \texttt{msg\_ack\texttt{(segment, }i\texttt{)}} to requesting DSMC;
      endif;
    \(V:\)
      increment synchronization variable;
      if variable value less than or equal to zero then
        remove an entry from queue\texttt{(segment, }i\texttt{)};
      endif;
      case entry of
        remote DSMC:
         send \texttt{msg\_ack\texttt{(segment, }i\texttt{)}} to remote DSMC;
         enqueue process on host-list; /* removed entry */
      endcase;
    endcase;
  endif;

5.3.3 Merging Data Transfer and Locking

Our data transfer primitives eliminate invalidation messages by keeping exactly one copy of a read-write segment. However,
keeping just one copy reduces availability (for readers) in applications that can be modeled as a readers/writers problem. The fact that there is exactly one copy of a read-write segment is appropriate from the point of view of the writers while being a severe restriction for the readers. Since we expect such applications to be encountered quite frequently, we propose the following modification to the get primitive to increase the availability. The modification is to include mode information in the primitive: get(object, segment, mode), where mode can be one of read-only, read-write, or none. The DSMC keeps two additional pieces of information in the segment table for read-write segments: lock mode, and readers. When the owner DSMC receives a request for a read-write segment it does the following:

```c
if queue(segment) empty then
    if (lock-mode = none) and (mode = none) then
        case keeper of
            self:
                if not memory(segment) then
                    bring segment into object-memory from disk;
                    endif;
                change keeper to requesting DSMC;
                send msg_segment(segment) to requesting DSMC;
                remote:
                    send msg_forward(segment) to keeper;
                    /* note at most one forwarding */
                    change keeper to requesting DSMC;
                    endcase;
                return;
            else /* mode <> none */
                enqueue requesting DSMC on queue(segment);
                return;
        endif;
    else /* mode <> none */
        if keeper is remote then
            /*
             * the owner remains the keeper when get requests are issued
             * with mode = read-only or mode = read-write;
             * if keeper is remote, he acquired the segment in mode 'none';
             * so grab the segment back from him */
            send msg_get(segment, none) to keeper;
            receive msg_segment(segment) from keeper;
            change keeper to self;
            endif;
        endif;
    endif;
if (lock-mode <> read-write) and (mode = read-only) then
    /* readers can enter if lock-mode is either read-only or none */
    set lock-mode to read-only;
    increment readers;
    send msg_segment(segment) to requesting DSMC;
else (lock-mode = none) and (mode = read-write) then
    /* a writer can enter if lock-mode is none */
    set lock-mode to read-write;
    send msg_segment(segment) to requesting DSMC;
else
    enqueue requesting DSMC on queue(segment);
    endif; /* if (lock-mode <> read-write)... */
else /* queue is non-empty */
    enqueue requesting DSMC on queue(segment);
    endif; /* if queue(segment)... */
endif;
```

The algorithm does not result in the generation of any invalidation messages. The readers (and writers) have to explicitly send a discard message on the segment. The msg_discard message is used by the owner to release the read-write lock on the segment. On receiving a msg_discard request the DSMC does the following:

```c
if lock-mode = read-only then
decrement readers;
if readers = 0 then
    lock-mode = none;
endif;
else
    lock-mode = none;
endif;
process_queue:
if queue(segment) non-empty then
    if lock-mode = none then
        if first-entry.mode = read-only then
            for each entry in the queue
                send msg_segment(segment) to requesting DSMC;
            until (subsequent-entry.mode = read-write) or
                (queue(segment) is empty) or (subsequent-entry.mode = none);
        elseif (first-entry.mode = none) and
            (queue(segment) not empty) then
            send msg_error(segment, mode_conflict) to requesting DSMC;
            goto process_queue;
        elseif (first-entry.mode = none) and
            (queue(segment) is empty) then
            send msg_segment(segment) to requesting DSMC;
        else /* first-entry.mode = read-write */
            set lock-mode to read-write;
            send msg_segment(segment) to requesting DSMC;
        endif;
    endif;
endif;
```

Note that mode none is a programming anomaly amidst read-only and read-write requests, and is treated as such by responding with a msg_error message. Upon receiving the msg_error message, the initial requester may decide to re-issue the get request.

How does the kernel know when to discard a segment? In readers/writers problem the reader (or the writer) explicitly acquires the appropriate lock, reads (or writes), and releases the lock. In Clouds, lock and unlock are system operations. When the Clouds application programmer uses these primitives, the operating system translates these primitives to get and discard the appropriate segments, respectively. We note that there are possibilities of deadlock if the application fails to release a lock. If an application fails to perform an unlock operation, the algorithm is in error. We envision a deadlock detection mechanism in the operating system that is layered on top of the DSMC mechanisms. The deadlock detector would issue unlock requests on behalf of the deadlocked processes. Note that if the kernel fails to perform a discard operation upon an unlock request, it is tantamount to a kernel bug.

With this enhancement our DSMC provides as much as availability as Li's algorithms [24] without incurring the high cost of invalidation messages. Note that explicit discards from the readers are in lieu of the invalidation messages. However, we contend that these are exactly the minimum number of messages required to maintain a consistent shared data structure.

5.3.4 Weaker Semantics

The above DSMC primitives provide strong memory coherence, where a write to a location by a process is seen by other processes when they access the same location. While this criterion is appropriate for applications that rely on the DSMC to enforce memory coherence, other applications may find these primitives too restrictive. Some applications may need to provide for memory coherence as part of their algorithms, while other applica-
tions may not need strong memory coherence. For example, a system monitoring facility may need to inspect the contents of some segments without acquiring locks, while a distributed game that maintains the state of a graphics screen in shared memory may sacrifice strong memory coherence for better performance.

For such applications, we provide a simple mechanism to acquire a copy of a segment without enforcing memory coherence. We define weak-read mode for the get request. A get(segment, weak-read) request for a segment acquires a copy of the segment from the owner DSMC. Upon receiving a get(segment, weak-read) request, the owner DSMC sends a copy of the segment to the requester, regardless of the fact whether a copy of the segment exists in any other node.

5.3.5 Summary of Modes

In summary, using the get primitive a segment may be acquired in one of four modes: read-only, read-write, weak-read, or none. Read-only mode signifies non-exclusive access but guarantees that the segment will not change until the node explicitly discards the segment. Read-write mode signifies exclusive access (for the node) with a guarantee that the segment will not be thrown away until the node explicitly discards the segment. Weak-read mode signifies non-exclusive access with no guarantee whether the segment will change or not. None mode signifies exclusive access with no guarantee whether the segment will be taken away or not.

6 Implementation of DSMC

Ra [5,7] is an operating system kernel designed to be the nucleus of Clouds operating system. It is currently implemented on the Sun-3 architecture. Ra defines and manages three primitive abstractions: segment, virtual space, and isiba. The contents of a segment may only be accessed when that segment is mapped to a range of addresses in a virtual space. Virtual spaces abstract the notion of an addressing domain, and they are composed of segments. Ra isibas are an abstraction of the fundamental notion of computation or activity and can be thought of as lightweight processes.

The Ra kernel is responsible for mapping segments into virtual memory using the memory management hardware provided by the underlying architecture. The size of a segment is a multiple of the physical page size. Ra assumes the existence of partitions that are responsible for realizing, maintaining, and storing segments. Partitions are an example of system objects. System objects encapsulate necessary and/or useful operating system services and resource managers that have direct access to the Ra kernel, but are nonetheless outside the kernel. System objects include device drivers, resource managers, and user-level object support. Each partition provides (at least) the following calls for use by Ra: activate/deactivate segment, create/destroy segment, and page-in/page-out portions of segments. When Ra is instructed to service a segment request (e.g. to map a segment into a virtual space), it invokes the appropriate partition to fetch the segment into physical memory. Ra then manipulates the memory management hardware to map the physical pages appropriately.

We have implemented the DSMC as a software module that consists of approximately 3500 lines of C++ [34]. Figure 2 shows the organization of the DSMC implementation on Ra. The boxes in the figure denote system objects. The DSMC cooperates with remote DSMC's to implement the distributed shared memory primitives. DSM Partition is a Ra partition

\[\text{Figure 2: Organization of DSMC implementation under Ra}\]

that provides the kernel with the ability to create/destroy and activate/deactivate segments, page-in/page-out portions of segments, and semaphore P/V operations. The DSM partition decides if a segment is owned by the local node or a remote node. It uses the Disk Partition to access local segments, and cooperates with the DSMC to access remote segments. Transaction abstraction layer (TAL) is a simple transaction-oriented communication protocol that provides reliable multi-packet request/response messages. It is used by the DSMC to communicate with remote DSMC's. TAL protocol is similar to other transaction-oriented protocols such as VMTP [12]. However, it is much simpler than VMTP since it is tailored to our application domain. Because the DSMC algorithms require simple request/response messages only (possibly with message forwarding), it is possible to substitute other transaction-oriented protocols for TAL. The Disk Partition maintains segments owned by the local node on the local secondary storage (if any).

The DSMC implementation executes on top of Unix (as a user process) or Ra. The operating system dependencies are isolated in a few C++ classes. The organization of the DSMC implementation on Unix is shown in Figure 3. On Unix, TAL runs as a user process that uses SUN's Network Interface Tap (NIT) [35] to receive packets from the net and to route them

\[\text{Figure 3: Organization of DSMC implementation on Unix}\]
among a set of clients and servers. The DSMC code is linked-in with client code. DSMC code is also linked-in with server code that uses the Unix file system to store segments. Implementation of DSMC on Unix and Ra serves three purposes:

1. The Unix environment makes it easy to test and verify the DSMC and TAL protocols.
2. The Unix file system is available for use as permanent store for segments. Ra executes on diskless Sun-3 workstations with backing store provided by Unix machines.
3. The strength of Unix is the rich program development environment that it provides. The strength of Clouds is transparent management of distributed data and computation. Providing inter-operability between Unix and Clouds is one of our design goals. DSMC implementation on Unix and Ra serves this purpose. System and user objects are developed on Unix and demand-paged to Ra via DSMC mechanisms.

The Unix implementation of the DSMC and TAL is complete, and we report on its performance in §7. The Ra implementation of the DSMC is awaiting the completion of an ethernet driver for Ra.

7 Performance Results

We measured the performance of the DSMC implementation on Unix. All measurements are done on Sun-3/60 workstations with 4M bytes of memory, connected through a 10M bits/sec ethernet. We mark out the cost of secondary storage access by caching segments in memory before measuring the costs of the DSMC primitives. Table 1 summarizes the results.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ms)</th>
<th>Throughput (Kbytes/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get or discard (8K bytes)</td>
<td>43.4</td>
<td>185</td>
</tr>
<tr>
<td>with forwarding</td>
<td></td>
<td></td>
</tr>
<tr>
<td>throughput:</td>
<td>63.7</td>
<td>126</td>
</tr>
<tr>
<td>V operation:</td>
<td>16.5</td>
<td></td>
</tr>
<tr>
<td>P operation:</td>
<td>31.5</td>
<td></td>
</tr>
<tr>
<td>Activate segment:</td>
<td>23.3</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Measurements of DSMC operations on Unix

The Table shows that on an average fetching a segment (without forwarding) of size 8K bytes (the page size on the Sun-3) takes 43.4 ms. Van Renesse et al. report a transfer rate of 40 ms for 8K bytes between two user processes on different nodes using Sun RPC on a 10M bits/sec Ethernet [36]. Our implementation uses two user processes per node and still compares favorably with the figures reported by van Renesse et al. A null message from one DSMC to another costs roughly 20 ms, a large portion of which is spent context switching between the kernel and TAL, and between TAL and DSMC. Moving TAL into the Unix kernel would eliminate the additional context switching, and we are currently investigating such an implementation. A semaphore V operation costs only 16.5 ms since it is non-blocking, i.e., the issuing process continues without waiting for the final acknowledgment from the remote DSMC.

We have also evaluated our DSMC scheme and compared it to RPC using simulation [28]. In the simulation, the DSMC is treated as a software module as described in §6. The performance study shows that:

- Over a range of object invocation locality, DSMC has significant advantages over RPC.
- Use of distributed shared memory achieves automatic distribution of processing load, by performing the invocation locally instead of on the owner nodes.
- Instead of addressing memory coherency and synchronization separately, applications that can be modeled as readers/writers problems benefit considerably when locking and segment access primitives are combined.

As mentioned before, the DSMC primitives are provided to the operating system as a set of mechanisms for managing memory in the network. Policy decisions, such as when to use RPC and when to use the DSMC primitives, are in the operating system. The operating system decides how and when to use the DSMC primitives. For example, during periods of high concurrent access to an object, the operating system may use the DSMC primitives to locate the object at one node, and then use RPCs to invoke the object thereby reducing data movement across the network.

Comparison with Li’s Scheme

To illustrate the advantage of combining mutual exclusion and consistency maintenance, we show how a readers/writers problem can be implemented using our scheme and Li’s scheme, and compare the number of messages generated in each case. Li’s basic scheme can be summarized as follows:

- On a read fault:
  1. Ask manager for page.
  2. Manager forwards request to owner.
  3. Owner sends copy of page to requester.

- On a write fault:
  1. Ask manager for page.
  2. Manager forwards request to owner.
  3. Owner sends page and copy-set to requester.
  4. Requester invalidates all copies in copy-set by sending a message to each node holding a read-copy of the page.

Suppose we want to program the following readers/writers problem: A segment that is accessed by a set of readers and writers resides on the manager node M, and each of the readers/writers runs on a different node (for simplicity, we assume that no reader/writer runs on the manager node). Each reader/writer computes for a while, then accesses the shared segment. It is clear that some mechanism to synchronize access to the segment is needed.

In our scheme, locks (see §5.3.3) can be used to solve this problem as follows:

Reader:

loop
compute for a while
lock segment in read-only mode
/* lock generates one message to manager */
/* manager eventually sends back segment */
access segment
unlock segment
/* unlock generates one message to manager */
endloop

II-167
Similarly for a writer:

Writer:

\[
\text{loop} \\
\text{compute for a while} \\
\text{lock segment in read-write mode} \\
/\ast \text{lock generates one message to manager }/ \ast \\
/\ast \text{manager eventually sends back segment }/ \ast \\
\text{access segment} \\
\text{unlock segment sending modified segment back to manager} \\
/\ast \text{unlock generates one message to manager }/ \ast \\
\text{endloop}
\]

It is not clear how one can program this simple readers/writers problem using Li’s primitives, for Li does not address the issue of process synchronization. Process synchronization has to be addressed separately, because there is no way within Li’s scheme for a user to lock a page while accessing it. Therefore, we assume that a lock operation is implemented by sending a message to a distinguished server, requesting access to the shared segment in the required mode. When the reply is received from the server, the shared segment is accessed. When the reader/writer is finished accessing the segment, it sends an unlock message to the server.

Using our scheme, the lock operation generates 2 messages: one message to the manager, and another from the manager to the requester (the second message includes the segment). The unlock operation generates one message to the manager. The lock operation generates no messages, because the segment is made available locally as a result of the lock operation. Using Li’s scheme, the lock operation generates 2 messages: one message to the server, and another from the server back to the requester (indicating that the requester can now access the segment). The unlock operation generates one message to the server. However, accessing the segment may generate several messages. On a read-fault, 2 messages are generated: one to the current holder of the segment, and another from the holder to the requester (the second message includes the segment). In addition to the 2 messages required to bring the segment from the current holder to the requester, a write-fault also generates \( O(r) \) invalidation messages, where \( r \) is the number of readers. The invalidation messages are required because process synchronization is separated from consistency maintenance—the fact that the lock server gave permission for access to the segment is unknown to the consistency maintenance algorithm. The table below summarizes the comparison in terms of the number of messages:

<table>
<thead>
<tr>
<th>Number of messages</th>
<th>Reader</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our Scheme</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Li’s Scheme</td>
<td>5</td>
<td>( 5+O(r) )</td>
</tr>
</tbody>
</table>

8 Conclusion

The abstraction of distributed shared memory is attractive from the point of view of object-based systems such as Clouds as well as other software architectures that use the shared memory paradigm for process communication. Such architectures would benefit considerably if the underlying hardware were to provide some transparent mechanism for efficient access and consistency maintenance of the distributed shared memory. We presented an organization and mechanisms for supporting this abstraction. The novel features of our approach are the use of distributed shared memory as an alternative to RPC, and the exploitation of process synchronization to simplify consistency maintenance.

References


A Measurement-based Study of Hardware Support for Object Invocation*

Umakishore Ramachandran  M. Yousef A. Khalidi

Technical Report GIT-ICS-88/21
May 1988

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Ga 30332-0280 USA

Abstract

The object invocation paradigm is attractive for structuring distributed systems. Distributed object-based operating systems view the resources of the system as a collection of objects. Object invocation is the primary mechanism in such systems and is often used as a yardstick for measuring the system performance. However, existing systems of this flavor exhibit poor performance due to the mismatch between the requirements of the object invocation mechanism and the machine architecture. Through measurements of an existing object-based kernel, we present a breakdown of the costs involved in implementing the object invocation mechanism. The measurements suggest architectural solutions to improve the performance of such systems. We present our preliminary studies towards providing hardware support for the object invocation mechanism.

*This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.
Introduction

This paper explores architectural support to improve the performance of object-based distributed operating systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network. There are one or more processors and a certain amount of memory in each node. Nodes do not share memory; message exchange across the network is the only mechanism for communication between them.

Operating systems structures for a distributed environment follow one of two paradigms: message-passing or object-invocation. Message-based operating systems [1,2,3,4] place a message-passing kernel on each node, supporting processes and communication between them via explicit messages. This kernel supports both local communication—communication between processes on the same node—and non-local or remote communication, sometimes implemented via a distinguished network manager process. In a traditional system such as Unix, access to system services is requested via protected procedure calls, whereas in a message-based operating system it is requested via message passing. Message-based operating systems are attractive for structuring distributed systems due to the separation of policy, encoded in server processes, from mechanism in the kernel.

Object-based distributed operating systems [5,6,7,8] view the resources of the system as a collection of objects. Objects are similar to abstract data types, and are written as individual modules composed of the specific operations that define their interface [8]. Access to system services is requested by invocation on the appropriate system object. In this sense, object-based distributed operating systems combine the advantages of both the traditional systems and the message-based systems. The invocation mechanism is similar to a protected procedure call, while objects encapsulate functionality similar to the server processes of message-based systems. Object invocation is the fundamental facility in object-based systems, and the speed of object invocation is often used as a yardstick for measuring the performance of such systems. The objective of this research is to understand the costs incurred in object invocation, and to propose hardware and software mechanisms to reduce these costs and hence improve the performance of object-based systems.

Clouds [5] is used as the testbed for evaluating our research ideas. While the issues investigated in this research are presented in the context of Clouds, they are general and apply to other object-based systems.

Clouds is an object-based distributed operating system being developed at Georgia Tech.
Objects and threads are the basic building blocks in Clouds. Objects are passive and persistent entities in the system. An object is the encapsulation of the code and data structures needed to implement the entry points in the object. These entry points provide the procedural interface for an activity to execute the code in an object. This code may itself call entry points in the same or other objects. Objects are disjoint partitions of a global virtual space that spans the entire network. Objects consist of one or more of the following types of areas: read-only code, read-only data, shared read-write data. These areas are referred to as segments.

Threads are the only active entities in the system. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object. A thread enters an object by invoking an entry point in the object. It then executes the code in the entry point, and returns to the caller object. The binding of an object invocation to an entry point in an object takes place at execution time, and more than one invocation may execute in the same object concurrently. Figure 1 shows the model of computation in Clouds. A thread is comparable to a process as defined in many conventional systems, with the exception that a thread may span machine boundaries. For the purposes of this paper, a process is synonymous to a thread.

Measurements

A performance study of several object-based operating systems is desirable to ensure that we are not discovering coding inefficiencies of one operating system, but instead see a trend that is common to all systems. Unfortunately, there are very few systems, implemented on bare hardware, that can be helpful in such a study. Alpha [8] and Clouds [9] are appropriate systems to study, but we only have access to the Clouds kernel. Therefore, the Clouds kernel is used as the system to study, but necessary care must be taken so as not to draw the wrong conclusions when analyzing the Clouds implementation. The implementation may hide the real costs of object invocation. Therefore, a two-step strategy is followed:

- Extensive profiling of the current implementation is done. Implementation inefficiencies are identified, and modifications to the kernel are made to remove these inefficiencies.

- The modified kernel is then analyzed, and costs intrinsic to object-based systems implemented on conventional machines are identified.

To identify and eliminate all inefficiencies amounts to rewriting the kernel. Therefore, an effort is made to remove glaring inefficiencies and to conduct the experiments in such a manner as to
exercise only the portions of the kernel that we are interested in. For example, by bringing in objects into main memory before starting an experiment, the performance of secondary storage is masked out. It is to be noted that a new implementation is currently underway based on our experience with the existing system. The new implementation is discussed in a subsequent section.

Overview of the invocation mechanism in Clouds

Local invocation in passive-object systems involves mapping the required memory segments of the object into the address space of the invoking thread. Remote invocation is implemented as a local invocation at the remote node where the invoked object resides, similar to a remote procedure call [10].

The Clouds kernel is implemented on the Vax-11 architecture. The Vax architecture provides paged virtual memory with the system kernel mapped into one half of the memory space (system space), and the currently executing process mapped into the other half (process space). The process space is further divided into the Vax designated P0 and P1 spaces. Program images and most of their data reside in P0 space, while P1 space contains the process stacks and other data [11]. A page table that resides in physical memory describes the system space, and two separate page tables describe the currently mapped process space. The process page tables reside in system space virtual memory. The hardware provides a translation look-aside buffer (TLB) for caching recent virtual-to-physical memory translations. The TLB is split into two halves: one half caches translations from the system space, while the other half caches translations from the process space. The latter part is invalidated on each process context switch.

Local object invocation is considered first. When a thread $t$ executing in object $O_1$ invokes an entry point in object $O_2$, the following steps are taken: the calling object $O_1$ constructs two argument lists (arglists), one for transferring arguments out, and the other to receive output parameters from object $O_2$. Each arglist consists of a count field, an overflow indicator field, and an array of argument descriptor records. Each data descriptor record specifies one argument or a pointer to an argument (see Reference [12] for more details). After constructing the arglists, thread $t$ enters the kernel through a protected system call (trap). The parameters to the system call include the starting addresses of the two arglists, a capability to object $O_2$, and the number of the entry point to invoke in $O_2$. The kernel then calls the ObjInvoke routine that uses the capability to search an in-memory system table for $O_2$. If a descriptor for $O_2$ is not found, local secondary storage is searched. When $O_2$ is found, its descriptor is read into memory, and
the information contained in the object's descriptor is used to construct a PO page table for
the object. The kernel then copies the arguments onto the process stack, allocates an ObjRec
structure that holds information about \( O_1 \), and links it to the \textit{process control block} (PCB) in a
LIFO manner to be used later during object return. A new mapping of the PO space is set up,
and a process context switch that actually maps \( O_2 \) into PO is executed. A side effect of the
context switch is the invalidation of the process' TLB. Execution continues in the ObjInvoke
routine which returns to the trap handling routine, which in turn "returns" to object \( O_2 \) to start
the invocation.

Before \( O_2 \) returns to \( O_1 \), an arglist of return parameters is constructed. The thread \( t \) then
enters the kernel through the protected system call interface, passing back the starting address
of the return arglist and a success/error indication. The kernel calls ObjReturn to perform the
following steps: the first ObjRec is dequeued from the PCB, and the information contained in it
is used to locate \( O_1 \). The return parameters are copied into a temporary area, a new mapping
of the PO space is set up, and a context switch is executed. The return parameters are then
copied into the locations specified by the "out" arglist. ObjReturn then returns to \( O_1 \), via the
trap handling code.

If \( O_2 \) is not found locally when thread \( t \) enters the kernel during object call, \( O_2 \) is assumed to
be remote and an invocation request is broadcast to other nodes. The RPC server on the node
that has \( O_2 \) acknowledges the invocation request and creates a local slave process to invoke \( O_2 \) on
behalf of \( t \). The slave process proceeds to invoke \( O_2 \) locally, and when the invocation completes,
it sends the return arglist back to the invoking node. When the return arglist is received, it is
acknowledged, and ObjReturn is called to return to \( O_1 \). The two important differences between
local and remote invocation are the necessity to send messages through the local area network,
and the need to employ a slave process at the remote node to perform the invocation.

Measurements and improvements

We added a profiling facility to the Clouds kernel. The mechanism has two components:

- The program counter (PC) is sampled every clock tick (10ms). Each time the PC is
  sampled, a counter corresponding to the value of the PC is incremented. Each 4 bytes of
  kernel code has a corresponding counter, and a separate counter is incremented when the
  value of the PC falls in user space.
For each procedure call, the caller is noted, plus a count is maintained of the number of times the procedure has been called by this caller.

The PC sampling and the procedure call counts are collected during run-time. This information is then used to construct a graph of the dynamic call sequence. The dynamic call graph along with the symbol table and the static graph of the program are then used to produce a detailed analysis of the kernel during the time when profiling is enabled. For each procedure, the information produced includes the percentage of the total run time spent in it, the number of times it is called, and the percentage of the time spent in each of its children. The algorithms used to build the dynamic and static graphs, and to calculate the time spent in each procedure are based on the Unix utility `gprof`.

In the following experiments, the objects involved are brought in memory to factor out the secondary storage access, and the profiling is done on an otherwise idle system. The costs of occasional reception of broadcast messages from the Ethernet plus the cost of the profiling code itself are factored out of the results.

Local Invocation

To measure the costs involved in object invocation, two objects `Caller` and `Callee` are used in the following experiments. In each experiment, 5 separate runs are made, with object `Caller` calling an entry point in object `Callee` 10,000 times, transferring-in 0 to 4096 bytes of data. Object `Callee` transfers-out 0 to 2048 bytes. Table 1 lists the cost of each object invocation—the sum of kernel and user times. Tables 2 and 3 give a breakdown of the total object call time, for the two cases of 0 and 1024 bytes of transferred-in data, respectively. (In Tables 2 and 3, “misc” refers to housekeeping activities performed by the kernel, such as initializing the contents of the `ObjRec` structure.)

Allocating/deallocating the `ObjRec` structure and enabling/disabling interrupts account for 60.9% of the total kernel time (Table 2). This high cost is an artifact of two glaring inefficiencies in the current implementation:

- On each object call the system memory heap is accessed to allocate an `ObjRec` structure. On object return, the heap is called to release this structure.
- Each of the disable and enable routines, coded in assembly language, are called twice during object invocation. The objective of the disable routine is to raise the processor level while
saving state. The enable routine lowers the processor level only if the previous level is
lower than the current level. A counter and a variable that holds the saved processor level
are used to keep track of the nesting depth of enable/disable.

These inefficiencies can be eliminated with the following modifications:

- Allocating a free list of empty ObjRec structures, and accessing the heap only when the
  list is exhausted is expected to save most of the time spent in accessing the heap.

- The enable routine can be replaced with four in-line instructions that exchange the pro-
  cessor level with a new level and save the old level on the current stack. Similarly, the
disable routine can be replaced with three in-line instructions that set the processor level
to the previous level stored on the stack.

Table 4 shows the cost of object invocation after these two modifications to the kernel. Note
that the times shown in the table are the average cost per invocation, and include kernel and
user mode times. Tables 5 and 6 show the corresponding breakdown of the object invocation
time, for the two cases of 0 and 1024 bytes of transferred-in data, respectively. The time to
enable/disable interrupts is now an insignificant portion of the object invocation time and is
included as part of allocating the ObjRec structure in Tables 5 and 6. Comparing Tables 2
and 5, the total kernel mode time spent on object invocation is reduced by 45% from an average
of 2671 µsec to 1441 µsec.

Though more tuning seems possible, the decomposition of costs in Tables 5 and 6 are rep-
resentative of a typical object invocation mechanism implemented on a conventional machine.
Some of the costs shown in the Tables cannot be eliminated. For example, trap handling, check-
ing rights, and initializing the ObjRec structure consist of straight-line code that cannot be
eliminated. However, the measurements point to a mismatch between the invocation mecha-
nism and the machine architecture. This mismatch is discussed next.

**TLB flushing costs**

As mentioned before, two context switch operations are needed per object invocation. Each
context switch operation costs around 10%-18% of kernel mode object invocation time (see
Tables 5 and 6). Most of this cost is eliminated using a protected procedure call to effect object
invocation (see next section). However, there is still a hidden cost that results from switching
address spaces on invocation. This hidden cost is due to flushing the TLB, and it manifests itself mostly in user mode time. We now investigate this point in more detail.

Conventional memory management units (MMU) provide a process with one virtual address space. These MMUs typically have the following three components:

- An address space pointer that points to the address space mapping table, also referred to as the page table. Address space switching is usually accomplished with a hardware pointer update in most MMUs.

- A page table that holds mapping and protection information. In most MMUs the page tables are kept in physical memory. The page table may be organized as a linear table (e.g. Vax-11), a tree (e.g. Motorola 68851 [13], Sun-3 MMU [14]), or a hash table (e.g. IBM PC/RT [15], HP Precision Architecture [16]). The latter two organizations are aimed at minimizing the amount of physical memory space that has to be reserved for holding the page tables when the virtual memory space is large and sparse. Organization as a linear table necessitates the use of other mechanisms, such as paging the page tables themselves, to limit the size of the page tables in physical memory.

- A TLB that holds recent virtual to physical address translations. A TLB is a high-speed associative cache and its purpose is to eliminate the need to access page tables that are often kept in main memory on each virtual-to-physical translation. In many MMUs, TLB entries are invalidated on every address space switch.

Object-based systems such as Clouds present a problem that is not very well handled by conventional MMUs. A thread in the course of its computation traverses the virtual address spaces of several objects. The operating system assigns a unique address space for each object. When a thread invokes an entry point in an object two events occur:

- The memory management hardware switches the address space for the currently executing thread such that memory accesses henceforth will use the address space of the invoked object.

- The TLB is flushed due to the switch in address spaces.

When a thread returns to the caller object, these two steps are repeated to switch back to the address space of the caller object.
It is known that the frequency of context switches can affect the TLB performance significantly [17,18]. In conventional systems such as Unix, if the TLB size is large enough, adequate hit rate performance can be achieved by choosing an appropriate organization of the TLB [19,18]. However, our simulation studies [20,21] show that in object-based systems increasing the TLB beyond a certain size does not help in reducing the hit rate.

To illustrate the effects of flushing the TLB on each object call and return, an experiment is conducted on a modified version of the Clouds kernel. An object, named Same, with two entry points SameCaller and SameCallee is used. SameCaller calls SameCallee, accesses \( n \) pages of memory and repeats this sequence 10,000 times. SameCallee accesses \( n \) different pages of memory and returns to its caller. Each page in memory is 512 bytes in size. The Clouds kernel is modified such that an intra-object invocation does not result in a context switch. All the kernel steps involved in object invocation are performed, except that the actual context switch instructions are replaced with no-op instructions. Figure 2 compares the average cost of object invocation with and without TLB flushing. In the figure, \( t_T \) refers to the average total object invocation time, while \( t_u \) refers to the average time spent in user mode per object invocation. The total number of pages accessed in SameCaller or SameCallee per invocation is more than \( n \) because a number of code and stack pages are additionally accessed during each invocation.

On a context switch, the process TLB is flushed but the kernel TLB is left intact. After the context switch, almost all accesses to the process space (i.e. to thread data and to the newly installed object) are made in user mode. Therefore, the cost of TLB flushing manifests itself mostly in user mode time \( t_u \). There are, however, few pages in the process space that are accessed by the kernel after the context switch and before starting execution in the newly mapped object. On object call, some parameters (around 50 bytes) are pushed on the user stack (thus accessing at most 2 pages), and on object return the output parameters are copied into the user object. Therefore, TLB flushing affects mostly user mode time.

In Figure 2, comparing \( t_{T,\text{flush}} \) to \( t_{T,\text{noflush}} \) for the case of \( n = 2 \), shows that \( t_{T,\text{noflush}} \) is 17.9\% smaller than \( t_{T,\text{flush}} \). For \( n = 5 \), the difference is 19.4\%. In the next section, we discuss a TLB design that supports caching of recent address translations across object invocations.

Remote Invocation

To measure the cost of remote object invocation, we use the Caller and Callee objects again, but in a distributed setting, with each object loaded on a different machine. In these experiments, no data is transferred back from Callee to Caller. Caller invokes an entry point in Callee, the
processor on which Caller executes remains idle until the invocation returns, and the invocation is repeated 1000 times. Column R1 in Table 7 shows the total cost per object invocation for a range of transferred data. Tables 8 and 9 provide a rough breakdown of the costs after eliminating CPU idle time. The following observations can be made based on these measurements:

- As mentioned before, a slave process is created on the remote node to perform the invocation locally on behalf of the invoking thread. Table 8 shows that around 43% of the remote node CPU time is used in reclaiming slave processes and returning them to the pool of available slaves. Though no other activity is running on the machines during the experiment, the CPU idles only 23% of the total experiment time.

- The general mechanism of the heap is used by the RPC and networking code to allocate memory buffers. We replaced the kernel heap with local free lists of memory blocks of the required sizes. We also modified the enable/disable routines as previously described in local invocation. These modifications are included in the following two experiments, shown in columns R2 and R3 in Table 7, and they reduce remote invocation time by approximately 9%.

To illustrate how slave reclamation affects system behavior, the same remote invocation experiment is repeated without reclaiming slaves, i.e. assuming process reclamation costs zero time. The results are shown in column R2 in Table 7. The objective is to illustrate the effect of slave reclamation on RPC cost. Note that the difference between R1 and R2, when transferring 1024 bytes, is 27.02 ms. Of this time, 21.31 ms are due to slave reclamation costs and the rest are due to the modifications to buffer allocation and the enable/disable routines. The results of eliminating the two acknowledgments [22] from the RPC protocol are shown in column R3 in Table 7. From these experiments, the following observations are made concerning remote object invocation:

- Slave start/reclamation is very important and must be done fast. In the R1 case above, reclaiming slave processes is a bottleneck that slows both the local and remote nodes. While servicing remote invocation requests, the Clouds kernel is also trying to prepare the "dead" slaves for more work. Though the Clouds implementation of slave reclamation could be improved, there is always some cost for assigning cohort processes to perform remote invocations.

- The Clouds kernel passes little additional data on an object invocation, though a sophisticated operating system is expected to pass more thread-specific data. This information
may include the thread identifier name, accounting information, controlling terminal information, etc. The operating system would then use this thread-specific information to create an environment similar to the invoking thread's environment. Passing additional data and creating a new thread environment on the remote node is expected to add more time to remote procedure calls.

Thus, the additional cost in a remote (non-local) invocation involves (a) sending messages across the network, (b) setting up an environment similar to the invoking thread’s environment, and (c) assigning a cohort process to do the actual work. An alternative to sending the computation to a remote node, is to bring the required data to the local node. This alternative, which we call distributed shard memory, has been explored in other systems such as Emerald [23] and Apollo Domain [24]. In a subsequent section, the mechanisms needed to support distributed shared memory are presented, and a hardware module to assist in this function is proposed.

Memory Management Support

Based on our measurements, we believe that object-based systems would benefit considerably if the machine architecture were to provide support for a process to traverse and cache recent address translations in multiple address spaces. In many MMUs, a change in address-space pointer also results in flushing the TLB. For example, the Vax load process context instruction changes the page table pointers of process space and flushes the process space half of the TLB. Some MMUs, such as the Motorola 68851, the Amdahl 470V/7, and the IBM 3033, associate a virtual address tag with the TLB entries allowing entries for more than one process to be in the TLB at the same time. In the Motorola 68851 [13], a process identifier is stored in the TLB as part of the tag. Therefore, apart from the usual replacement of entries when the TLB is full, only when a process identifier is re-used the entries corresponding to that process identifier need to be flushed from the TLB. However, these MMUs do not meet the requirements of object-based systems as will be evident from the discussion at the end of this section.

To better support object invocation, a scheme is presented in which the address space switch is effected by a protected procedure call mechanism, and virtual-to-physical address translations are cached across object invocations [20]. This scheme, shown in Figure 3, is an extension of the one used in the Motorola 68851. The machine virtual address space is partitioned into three regions: the K space or kernel space; the P space or process space, where the currently running process has its stack and other process-related data; and the O space or object space, where the currently invoked object resides. Virtual memory is organized in pages. Our proposed MMU
uses a page table in physical memory to translate virtual page numbers, VPN, to physical page numbers, PPN. In the description to follow, “address space pointer” refers to the base of the page table for a particular address space.

Three base registers, KBR, PBR, and OBR, point to address spaces K, P and O, respectively. An object table, OT, holds the address space pointers of recently invoked objects. Every invocation results in the following:

- OBR is set to point to the address space of the invoked object by the operating system.
- The MMU searches OT for an entry with the same value as OBR. If such an entry is found, the index of this entry is used as an object identifier, and a current object register, COR in the MMU is set to this value. If no matching entry is found in the OT, an entry is re-used, and all TLB entries with the same object identifier as the index of the chosen entry are flushed.

A return from an object invocation is similar, with the operating system setting the OBR to the address space of the caller object. As is evident from this description, with our MMU the invocation is effected by a protected procedure call instead of a context switch.

The TLB entries are tagged with <object identifier, VPN> for O space translations. Translations in the P and K spaces are cached in the TLB with <CP, VPN> and <CK, VPN> as the tags respectively, where CP and CK are MMU constants outside the range of OT indices. The P space entries (with CP tag) are flushed on every process switch.

Virtual addresses generated by the CPU are translated as follows: An address from the O space is prefixed with the contents of COR, and an associative search of the TLB is performed. A hit occurs when the <COR, VPN> pair matches the TLB tag. An address from the P or K space is prefixed with the appropriate MMU constant, either CP or CK, for TLB lookup. A hit occurs when the <Const, VPN> pair matches the TLB tag.

When an address space pointer is re-used (e.g. when an object is destroyed), system software instructs the MMU to flush any matching entry in the OT with the same contents as the address space pointer. The MMU, in turn, flushes any corresponding entries in the TLB.

Evaluation

We evaluated our scheme using trace-driven simulation, and compared its performance to the usual case of flushing the TLB on each object call and return (see Reference [20] for more
In performing this study, we were faced with the lack of traces of programs running on object-based systems. The current Clouds implementation runs only a few "toy" programs, and does not support instruction tracing.

The following approach was used to generate the input for our simulator. A number of programs written in the ‘C’ programming language, each of which consisted of several separately compiled modules, were executed under Unix on a Vax-11, and an address trace was generated for each program. The address traces were in turn used to drive a simulator that treated inter-module calls as object calls. Whenever a call (return) crossed from one module to another, it was treated as an object call (return). Intra-module calls were treated as normal procedure calls. Calls to library routines, as well as data accessed while executing inside an object were treated as part of that object’s virtual address space.

Modular programs tend to group related functions and data in the same module, and to pass data by value across modules. This behavior is very similar to what we expect to have on Clouds, with separate objects replacing program modules. The test programs used in the simulation follow this style of programming, with an average size of 500 lines of code per module, which is about the size of a Clouds object. We, therefore, believe that traces generated as explained above are appropriate for evaluating our scheme. The traces, however, lack kernel mode addresses, and therefore our simulation does not consider translation of K space addresses.

The results of our performance study is summarized below, where T1 refers to our scheme, while T0 refers to the usual case of flushing the TLB on object invocations:

- For a given TLB size, the miss ratio of T1 is consistently less than the miss ratio T0. For a TLB size of 128, the miss ratio of T1 is 100 times less than the miss ratio of T0.

- The T1 miss ratio decreases as the TLB size increases beyond the point when the T0 scheme miss ratio levels off. T1 uses the additional TLB entries to cache more address translations that may be used in the future, unlike T0 which flushes cached translations on each object call and return.

- Depending on ratio of $T_h/T_m$ (where $T_h$ is the hit cycle time, and $T_m$ is the miss cycle time), the overall percentage improvement of T1 over T0 ranges from around 15% to 30%.

The memory management unit that we presented is tailored to support object invocation. It is an engineering solution combining the features available in commercial MMUs. The strengths and weaknesses of commercial MMUs vis-a-vis the features required for an efficient implementation of Clouds are discussed below:
The ability to cache TLB information across object invocation. The cost of flushing TLB entries at object call/return is an implicit cost that affects mostly user mode time. Systems with one large virtual address space do not require flushing the TLB across object invocations. Examples of such systems include the IBM RT/PC, HP Precision Architecture, and SPUR machine [25].

The ability to implement an object call/return by performing a small number of operations on the MMU. The cost of required MMU operations is an explicit part of object invocation. This cost is quantified by the number of MMU registers, either in software or hardware, that need to be modified on each object call/return. In our proposed MMU, changing the OBR is the only operation that is required to effect an object call/return. A paper design of object invocation implementation on segmented single virtual address space systems, such as IBM RT/PC and HP Precision Architecture, reveals that only a small number of MMU operations is required. However, our experience in implementing the new Clouds kernel on the Sun-3 MMU [9] reveals that many MMU operations are needed on this machine.

The ability to represent sparse address spaces efficiently. Inefficient representation of sparse address spaces results in page tables that have large memory requirements and lengthy initialization time, relative to the size of address space actually allocated. As mentioned before, page tables organized as trees or inverted tables are suitable for representing sparse address spaces, whereas linear page tables are not.

The ability to share memory among address spaces. This feature is required to support sharing of segments among virtual spaces. Sharing segment in our proposed scheme and other MMUs with page-tables organized as a tree is easy. A segment is shared between virtual spaces by sharing a page table sub-tree between the different page tables. MMUs with inverted page tables (e.g. IBM RT/PC) and virtually-addressed data caches (e.g. SPUR) cannot have two different virtual addresses mapping into the same physical address (i.e. no aliasing). Sharing of memory between different spaces is possible only through hardware segment sharing, which imposes limitations on the number of segments per object and the way they can be shared between virtual spaces. Clouds advocates a model of programming with a large number of possibly small-sized segments. Any segment is potentially sharable among virtual spaces. Because sharing can only be at the hardware segment level, a hardware segment maps exactly one software segment. For example, in the IBM RT/PC the CPU generates 32-bit virtual addresses with the high-order 4 bits of the virtual address selecting one of sixteen segment registers. In a paper design of object
space implementation on the IBM PC/RT, the sixteen segments are allocated as follows: seven each for O and P spaces, and two for K space. Each hardware segment maps at most one software segment.

The HP Precision Architecture organizes its virtual space differently. Each virtual address is composed of a segment register and an offset. User software can change some of the segment registers without kernel assistance. Using such an organization, the number of software segments per virtual space is not constrained by the number of hardware registers. However, user software is responsible for loading the segment registers with hardware segment numbers before accessing a segment that is not already mapped. This organization complicates user software and may result in performance degradation if the object is composed of many segments. Moreover, the segment registers are loaded with hardware segment numbers that the kernel has to set up. The kernel has to communicate these number to user software when objects are loaded in memory, and has to make sure that no object still uses a hardware segment number that is about to be reused.

- **The ability to allocate/deallocate ranges of addresses easily.** The ability to allocate/deallocate ranges of addresses is related to sharing of memory segments among spaces, because ranges of addresses are allocated/deallocated at the segment level. The number of page table and MMU operations required to allocate/deallocate a range of addresses affects the cost of attaching and detaching [5] segments to virtual spaces. Some systems (e.g. Vax and Sun-3 MMU) require traversing a potentially large number of page-table entries to allocate or deallocate a range of addresses. Both IBM RT/PC and HP Precision Architecture adequately support this feature.

In summary, MMUs with a single large virtual space come closest to meeting the requirements of maintaining an object space. They do not require TLB flushing across object invocation, and may not require a large number of operations to implement object call/return. However, such MMUs either restrict the amount of sharing possible between virtual spaces, or introduce software complexities when sharing segments.

The requirements discussed so far are basic to managing object space. Systems such as Clouds [9] and Argus [7], support the notions of atomicity of computation and recoverability of data, sometimes referred to as transactions. This notion requires that memory segments be recoverable if a computation needs to be aborted. Software techniques such as shadowing and logging are usually employed to implement transactions [26]. However, it is possible to reduce the burden on the software by providing some mechanisms in the MMU for supporting
transactions. IBM RT/PC implements one such mechanism called transaction locking in its MMU [15]. In Reference [27], Chang and Mergen report on implementing a transaction system using these mechanisms. We are currently evaluating the efficacy of incorporating transaction support in our MMU design. It should be noted that such a mechanism can be easily added to our present design.

Distributed Shared Memory

In a distributed object-based system, the virtual address spaces of all objects can be viewed as constituting a global distributed shared memory. Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-mapping mechanism. Local object invocation involves mapping the required memory segments of the object into the address space of the invoking process by installing the object as the current O space with the process’s P space. The current trend in structuring distributed systems is to use a collection of diskless computational servers or workstations, and a few data servers or file servers. In such an environment, the code and data for the local invocation has to be paged-in from the data server. Further, for remote object invocation we have one of two choices: The first choice is to perform the computation at the node where the object resides through remote procedure call. The second choice is to make the invocation appear local by bringing in the segments required for the invocation. While the former has to be supported for immovable objects such as an object that reads disk blocks, the latter may be a better choice for movable objects. There are two reasons to support this belief:

- the principle of locality [28] that suggests an invocation or other invocations in the same object may be repeated
- the reduction in computational overhead due to the elimination of slave process management to support remote invocation at the node where the object resides [21].

In References [29] and [21] the concept of distributed shared memory as an alternative to RPC is proposed, the algorithms to maintain strong and weak memory consistency are presented, and the algorithms are evaluated using simulation. Each object is owned by one node, and segments of an object can be at other nodes temporarily. Each node has associated with it a “network cache”—a repository for recently accessed remote memory-segments and their owners (nodes). A node that caches a segment from a remote owner is called a keeper of the segment.
The organization (Figure 4) proposed inside each node of the network is the following: a host that executes distributed applications; a distributed shared memory controller (DSMC) together with the network interface assists the host in mapping memory segments, both local and remote, into the virtual address spaces of the application processes. There is a minimal kernel on the host that traps system calls, and virtual address translation faults. The DSMC is also in control of the network. The system memory is logically partitioned into two parts: One part, object memory, is for housing the segments of locally created objects; the other part, network cache, is for caching segments from remote objects. Conceptually there are two lists of process control blocks: the host-list that the host looks at to schedule runnable processes; the DSMC-list that the DSMC looks at to service memory segment requests. The host enqueues processes that fault on virtual address translation in the DSMC-list. The DSMC enqueues processes that have become runnable again after the fault service in the host-list.

The basic operations provided by the DSMC are get, and discard. The get operation is used to fetch a segment (or a part thereof) from its owner, while discard is used to return a segment to its owner. The DSMC provides synchronization primitives as separate P and V semaphore operations, or as combined access and lock operations using the get and discard primitives.

Using the get primitive a segment may be acquired in one of four modes: read-only, read-write, weak-read, or none. Read-only mode signifies non-exclusive access but guarantees that the segment will not change until the node explicitly discards the segment. Read-write mode signifies exclusive access (for the node) with a guarantee that the segment will not be thrown away until the node explicitly discards the segment. Weak-read mode signifies non-exclusive access with no guarantee whether the segment will change or not. None mode signifies exclusive access with no guarantee whether the segment will be thrown away or not.

We evaluated our DSMC scheme and compared it to RPC using simulation [29]. In the simulation, the DSMC is treated as a software module that is part of the kernel. The performance study shows that:

- Over a range of object invocation locality, DSMC has significant advantages over RPC.
- Use of distributed shared memory achieves automatic distribution of processing load, by performing the invocation locally instead of on the owner nodes.
- Instead of addressing memory coherency and synchronization separately, applications that can be modeled as readers/writers problems benefit considerably when locking and segment
access primitives are combined.

In Reference [5] the design and implementation of a native kernel that integrates distributed shared memory for network-wide memory management is discussed. The DSMC primitives are provided to the operating system as a set of mechanisms for managing memory in the network. Policy decisions, such as when to use the RPC mechanism and when to use the DSMC primitives, are in the operating system. For example, during periods of high concurrent access to an object, the operating system uses the DSMC primitives to locate the object at one node, and uses RPCs to invoke the object thereby reducing data movement across the network.

Our work is built on the large body of work that exists in maintaining cache coherence in multiprocessors [30]. The DSMC algorithms deal with providing the mechanisms needed to maintain consistency of shared data in a non-shared memory architecture. Kai Li in Reference [31] addresses the same problem, wherein the entire memory is composed of untyped pages that are potentially sharable for both reads and writes. The novelty in our work is in exploiting application semantics to type the segments as read-only or read-write, and using the type specifier in the DSMC primitives as hints for simplifying consistency maintenance. By merging process synchronization with data transfer, the DSMC primitives provide mutual exclusion for free.

Other researchers have proposed the use of shared memory as a distributed systems structuring concept [32]. Emerald is a distributed object-based language and system with support for object mobility [23]. The Apollo Domain system [24] is a loosely-coupled network of computers that provide the user with a view of a single-level store. This view allows programs to share files, specifying the semantics of sharing, such as exclusive/non-exclusive, at the time of opening the file. On opening the file, it is mapped into the virtual address space of the program, and henceforth reads and writes to the file are no different from simple memory reads and writes. These system architectures assume the existence underneath of a consistency preserving mechanism similar to Li's and ours. Distributed shared memory in an object-based environment plays a role similar to the one played by network file systems, such as NFS [33] and Sprite [34], in conventional systems.

Conclusions and Future Work

The Clouds kernel served well as a prototype, and enabled us to gain important insight into the requirements of the object-based model of computation. However, it has several drawbacks. It is a monolithic kernel that is hard to modify and maintain, and the implementation is very
machine-dependent. In addition, message transmission in Clouds is slow. Sending messages from one process running in a local kernel to a process running in a remote kernel takes roughly 10 ms. Most of this cost can be attributed to a poor network interface [35]. Other systems, such as QuickSilver [3] and V [1], report numbers in the range of 6 to 1.5 ms on faster hardware.

Our notions on structuring object-based operating systems has matured since the prototype design was begun. A new kernel for Clouds, called the Ra kernel, has been designed and implemented [5,9]. Ra runs on the Sun-3 architecture, and incorporates support for distributed shared memory. The DSMC has been implemented in software and we are currently in the process of evaluating the implementation [36,37]. Further refinement and implementation of an MMU tailored to object-based systems that incorporates our TLB scheme, and a hardware module that implements the DSMC protocol are some of the work we have identified for future research.
References


Tevanian, and Michael Young. Mach: a new kernel foundation for Unix development. In *


kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 64–75,
December 1981.

The architecture of Ra: a kernel for Clouds. In *Proceedings of the Twenty-Second


[9] José M. Bernabéu Aubán, Phillip W. Hutto, M. Yousef A. Khalidi, Mustaque Ahamad,
Clouds — a distributed, object-based operating system: architecture and kernel
implementation. In *European UNIX systems User Group Autumn Conference, EUUG, Oc-
tober 1988.*

[10] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *


of Information and Computer Science, Georgia Institute of Technology, 1986. Available as


800-1780-10.


---

**Figure 1: Model of Computation**
Figure 2: Comparison of TLB flushing vs. no flushing.
Figure 3: Proposed MMU Structure

Figure 4: Hardware Implementation
### Table 1: Local object invocation costs

<table>
<thead>
<tr>
<th>Bytes (in + out)</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.4910</td>
</tr>
<tr>
<td>32 + 16</td>
<td>3.6040</td>
</tr>
<tr>
<td>64 + 32</td>
<td>3.6430</td>
</tr>
<tr>
<td>128 + 64</td>
<td>3.6710</td>
</tr>
<tr>
<td>256 + 128</td>
<td>3.7670</td>
</tr>
<tr>
<td>512 + 256</td>
<td>3.8770</td>
</tr>
<tr>
<td>1024 + 512</td>
<td>4.3020</td>
</tr>
<tr>
<td>2048 + 1024</td>
<td>5.1580</td>
</tr>
<tr>
<td>4096 + 2048</td>
<td>6.3220</td>
</tr>
</tbody>
</table>

### Table 2: Local object invocation, 0 bytes

<table>
<thead>
<tr>
<th>Description</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total kernel mode time</td>
<td>2671 usec</td>
</tr>
<tr>
<td>Trap handling</td>
<td>160 usec</td>
</tr>
<tr>
<td>ObjInvoke</td>
<td></td>
</tr>
<tr>
<td>Locating object in memory</td>
<td>97 usec</td>
</tr>
<tr>
<td>Checking rights + context switch + misc</td>
<td>265 usec</td>
</tr>
<tr>
<td>Processing Args</td>
<td>99 usec</td>
</tr>
<tr>
<td>Allocating the ObjRec structure</td>
<td>769 usec</td>
</tr>
<tr>
<td>Disabling and enabling interrupts</td>
<td>287 usec</td>
</tr>
<tr>
<td>ObjReturn</td>
<td></td>
</tr>
<tr>
<td>Context switch + misc</td>
<td>215 usec</td>
</tr>
<tr>
<td>Processing Args</td>
<td>176 usec</td>
</tr>
<tr>
<td>Deallocating ObjRec structure</td>
<td>285 usec</td>
</tr>
<tr>
<td>Disabling and enabling interrupts</td>
<td>287 usec</td>
</tr>
<tr>
<td>Other</td>
<td>31 usec</td>
</tr>
<tr>
<td>User mode time</td>
<td>820 usec</td>
</tr>
<tr>
<td></td>
<td>Time (µsec)</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Total kernel mode time</td>
<td>3374</td>
</tr>
<tr>
<td>Trap handling</td>
<td>160</td>
</tr>
<tr>
<td>ObjInvoke</td>
<td></td>
</tr>
<tr>
<td>Locating object</td>
<td>97</td>
</tr>
<tr>
<td>in memory</td>
<td></td>
</tr>
<tr>
<td>Checking rights</td>
<td>265</td>
</tr>
<tr>
<td>+ context switch</td>
<td></td>
</tr>
<tr>
<td>+ misc</td>
<td></td>
</tr>
<tr>
<td>Processing Args</td>
<td>463</td>
</tr>
<tr>
<td>Allocating the ObjRec</td>
<td>760</td>
</tr>
<tr>
<td>structure</td>
<td></td>
</tr>
<tr>
<td>Disabling and enabling</td>
<td>297</td>
</tr>
<tr>
<td>interrupts</td>
<td></td>
</tr>
<tr>
<td>ObjReturn</td>
<td></td>
</tr>
<tr>
<td>Context switch + misc</td>
<td>224</td>
</tr>
<tr>
<td>Processing Args</td>
<td>477</td>
</tr>
<tr>
<td>Deallocating ObjRec</td>
<td>290</td>
</tr>
<tr>
<td>structure</td>
<td></td>
</tr>
<tr>
<td>Disabling and enabling</td>
<td>297</td>
</tr>
<tr>
<td>interrupts</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>44</td>
</tr>
<tr>
<td>User mode time</td>
<td>928</td>
</tr>
</tbody>
</table>

Table 3: Local object invocation, 1024 bytes

<table>
<thead>
<tr>
<th>Bytes (in + out)</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.2610</td>
</tr>
<tr>
<td>32 + 16</td>
<td>2.2490</td>
</tr>
<tr>
<td>64 + 32</td>
<td>2.2900</td>
</tr>
<tr>
<td>128 + 64</td>
<td>2.3240</td>
</tr>
<tr>
<td>256 + 128</td>
<td>2.4040</td>
</tr>
<tr>
<td>512 + 256</td>
<td>2.5990</td>
</tr>
<tr>
<td>1024 + 512</td>
<td>3.0220</td>
</tr>
<tr>
<td>2048 + 1024</td>
<td>3.7190</td>
</tr>
<tr>
<td>4096 + 2048</td>
<td>5.5950</td>
</tr>
</tbody>
</table>

Table 4: Modified local object invocation costs
<table>
<thead>
<tr>
<th>Total kernel mode time</th>
<th>1441 usec</th>
<th>100.0%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trap handling</td>
<td>160 usec</td>
<td>11.1%</td>
</tr>
<tr>
<td>ObjInvoke</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Locating object in memory</td>
<td>97 usec</td>
<td>6.7%</td>
</tr>
<tr>
<td>Checking rights + context switch + misc</td>
<td>265 usec</td>
<td>18.4%</td>
</tr>
<tr>
<td>Processing Args</td>
<td>99 usec</td>
<td>6.9%</td>
</tr>
<tr>
<td>Allocating the ObjRec structure &amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disabling and enabling interrupts</td>
<td>199 usec</td>
<td>13.8%</td>
</tr>
<tr>
<td>ObjReturn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context switch + misc</td>
<td>215 usec</td>
<td>14.9%</td>
</tr>
<tr>
<td>Processing Args</td>
<td>176 usec</td>
<td>12.2%</td>
</tr>
<tr>
<td>Deallocating ObjRec structure &amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disabling and enabling interrupts</td>
<td>199 usec</td>
<td>13.8%</td>
</tr>
<tr>
<td>Other</td>
<td>31 usec</td>
<td>2.15%</td>
</tr>
<tr>
<td>User mode time</td>
<td>820 usec</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Modified local object invocation, 0 bytes

<table>
<thead>
<tr>
<th>Total kernel mode time</th>
<th>2094 usec</th>
<th>100.0%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trap handling</td>
<td>160 usec</td>
<td>7.6%</td>
</tr>
<tr>
<td>ObjInvoke</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Locating object in memory</td>
<td>97 usec</td>
<td>4.6%</td>
</tr>
<tr>
<td>Checking rights + context switch + misc</td>
<td>265 usec</td>
<td>12.7%</td>
</tr>
<tr>
<td>Processing Args</td>
<td>463 usec</td>
<td>22.0%</td>
</tr>
<tr>
<td>Allocating the ObjRec structure &amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disabling and enabling interrupts</td>
<td>182 usec</td>
<td>8.7%</td>
</tr>
<tr>
<td>ObjReturn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context switch + misc</td>
<td>224 usec</td>
<td>10.7%</td>
</tr>
<tr>
<td>Processing Args</td>
<td>477 usec</td>
<td>22.8%</td>
</tr>
<tr>
<td>Deallocating ObjRec structure &amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disabling and enabling interrupts</td>
<td>182 usec</td>
<td>8.7%</td>
</tr>
<tr>
<td>Other</td>
<td>44 usec</td>
<td>2.1%</td>
</tr>
<tr>
<td>User mode time</td>
<td>928 usec</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Modified local object invocation, 1024 bytes

26
Table 7: RPC costs in milliseconds

<table>
<thead>
<tr>
<th>Bytes (in)</th>
<th>RPC Method</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52.69</td>
<td>26.9</td>
<td>21.9</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>52.71</td>
<td>27.2</td>
<td>22.1</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>52.74</td>
<td>27.4</td>
<td>22.6</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>52.78</td>
<td>28.0</td>
<td>23.0</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>53.00</td>
<td>29.0</td>
<td>24.1</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>54.52</td>
<td>32.6</td>
<td>27.5</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>64.92</td>
<td>37.9</td>
<td>32.8</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>85.08</td>
<td>57.5</td>
<td>57.5</td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td>125.15</td>
<td>89.9</td>
<td>89.9</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Remote object invocation (R1), 1024 bytes, remote node

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time (including idle time)</td>
<td>64.92 ms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total utilized CPU time</td>
<td>49.94 ms</td>
<td>100.0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User mode time</td>
<td>0.60 ms</td>
<td>1.2%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel mode time</td>
<td>49.34 ms</td>
<td>98.8%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slave Reclamation</td>
<td>21.31 ms</td>
<td>42.7%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RPC server total time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time sending/receiving messages</td>
<td>9.96 ms</td>
<td>19.9%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Starting slave</td>
<td>3.67 ms</td>
<td>7.3%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Locating object in memory</td>
<td>0.94 ms</td>
<td>1.9%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RPC return</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(includes CPU time to send results back)</td>
<td>10.04 ms</td>
<td>20.1%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local object invocation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(object call + return)</td>
<td>2.20 ms</td>
<td>4.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miscellaneous overhead</td>
<td>1.22 ms</td>
<td>2.4%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Remote object invocation (R1), 1024 bytes, remote node
<table>
<thead>
<tr>
<th>Description</th>
<th>Time</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time (including idle time)</td>
<td>64.92 ms</td>
<td></td>
</tr>
<tr>
<td>Total utilized CPU time</td>
<td>13.21 ms</td>
<td>100.0%</td>
</tr>
<tr>
<td>User mode time</td>
<td>0.33 ms</td>
<td>2.5%</td>
</tr>
<tr>
<td>Kernel mode time</td>
<td>12.88 ms</td>
<td>97.5%</td>
</tr>
<tr>
<td>ObjInvoke</td>
<td>1.67 ms</td>
<td>12.6%</td>
</tr>
<tr>
<td>RPC_Invoke</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocating network ports &amp; housekeeping overhead</td>
<td>3.00 ms</td>
<td>22.7%</td>
</tr>
<tr>
<td>RPC_Send</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sending invocation request</td>
<td>3.10 ms</td>
<td>23.5%</td>
</tr>
<tr>
<td>Receiving acknowledgement</td>
<td>2.01 ms</td>
<td>15.2%</td>
</tr>
<tr>
<td>RPC_GetResults</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receiving results</td>
<td>2.05 ms</td>
<td>15.5%</td>
</tr>
<tr>
<td>Sending acknowledgement</td>
<td>0.93 ms</td>
<td>7.0%</td>
</tr>
<tr>
<td>Miscellaneous overhead</td>
<td>0.13 ms</td>
<td>1.0%</td>
</tr>
</tbody>
</table>

Table 9: Remote object invocation (R1), 1024 bytes, local node
List of Tables

1  Local object invocation costs .................................................. 24
2  Local object invocation, 0 bytes .............................................. 24
3  Local object invocation, 1024 bytes .......................................... 25
4  Modified local object invocation costs ......................................... 25
5  Modified local object invocation, 0 bytes .................................... 26
6  Modified local object invocation, 1024 bytes ................................ 26
7  RPC costs in milliseconds .......................................................... 27
8  Remote object invocation (R1), 1024 bytes, remote node .................. 27
9  Remote object invocation (R1), 1024 bytes, local node ..................... 28

List of Figures

1  Model of Computation ............................................................. 21
2  Comparison of TLB flushing vs. no flushing ................................... 22
3  Proposed MMU Structure .......................................................... 23
4  Hardware Implementation .......................................................... 23
Processor Initiated Sharing in Multiprocessor Caches *

Umakishore Ramachandran
Joonwon Lee

Technical Report GIT-ICS-88/43
November 1988

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332-0280 USA

rama@gatech.edu, joon@pyr.gatech.edu

Abstract

We present a lock-based multiprocessor cache consistency protocol. This protocol requires the processes to provide information about the data they access: read or write, shared or non-shared. This information enables caches to avoid frequent invalidations and data transfers. Shared data can be accessed only through lock operations. These lock operations entail queues for the processes awaiting a lock. A distributed hardware-assisted queue which ensures fairness and efficient waiting scheme is proposed. Implementation of the protocol with a current computer bus is discussed.

Key Words: Shared memory multiprocessor, Cache coherency, Process synchronization

*This work is supported in part by NSF grant MIP-8809268
1 Introduction

With the advent of shared memory multiprocessors, there has been a tremendous interest in the research community for developing algorithms to maintain the consistency among the data cached in the processors and the global memory (see [AB86] for a survey). Multiprocessor cache consistency algorithms ensure the following memory coherence constraint: a "read" operation performed by a processor returns the "most recent" value written into that location (by any processor). Given that processor clocks are not synchronized, does this coherence criterion make sense? This criterion is appropriate in a shared memory multiprocessor since the system bus (a broadcast medium) serializes the memory operations of all the processors. Most of these algorithms consider memory coherence problem in isolation. In reality, memory coherence and process synchronization are closely intertwined. The ability to read or write shared data (by a process) is invariably acquired through some synchronization method. Since the cached data has no synchronization information associated with it, these algorithms tend to be an overkill owing to their generality. Nevertheless, these protocols are a viable approach for solving the cache coherency problem in multiprocessors with a few processors since the total cost (measured in circuit complexity as well as time) of implementing them in hardware is a small fraction of the total system cost. However, for large-scale shared memory multiprocessors the increase in bus traffic due to false sharing [EK88] can be a limiting factor to the scalability of the machine. Further, since access to shared data is usually preceded by the acquisition of an appropriate lock, there is increased delay in access to shared data first to acquire the lock and then to acquire the actual data.

We propose a lock scheme based on snooping caches to merge process synchronization with the actual data transfer. For such a scheme to be practical, the cache controller should support an efficient waiting method to prevent processors from generating unsuccessful retries on the bus, while ensuring fairness to all waiting processes. A lock is either shared or exclusive. This distinction is usually provided by the software and is indispensable for optimal sharing of the data. A queue is an acceptable principle for a waiting scheme because it is simple to implement, easy to analyze, and most importantly, fair. We propose a distributed hardware-assisted queue since a centralized queue allows only serialized access defeating the purpose of snooping caches.
2 Related Work

There are two approaches to maintaining the coherence of multiprocessor caches: software controlled and hardware controlled. The former approach \cite{CV88,EGK85} relies on the system software (operating systems and compiler) to invalidate stale copies of a line in the cache, thus reducing the hardware complexity. This approach has been applied to general interconnection networks as well as shared buses. However, it requires sophisticated software and an efficient synchronization method to avoid invalidation of valid copies. The hardware controlled schemes allow any line to be cached by any processor and rely on cache controllers to keep the replicated lines coherent. Early work in this area was based on a centralized control \cite{YYF85,Tan76}, but recent research is focused on distributed control because it is easily expandable and enables a cache to respond quickly to requests.

Yen et. al. \cite{YYF85} describe a write-through scheme used in IBM 370/168 and IBM 3033. If a line is updated in a cache, it is written through to the memory and an invalidation signal is broadcast on the bus; other caches with a matching entry invalidate the line. Since a large portion of cache directory cycles are used to search for matching entries for invalidation the performance is significantly degraded as the write rate to the cache from a processor increases. Albeit the invalidation cycle can be reduced by a sophisticated filtering method, bus traffic to broadcast invalidation signal is still a considerable overhead. So, in this scheme, the bus is saturated very quickly as the number of processors increases \cite{YYF85}.

Goodman \cite{Goo83} describes a write-once scheme for use with Multibus. The first write to a line in cache goes through to the memory (one word write) to invalidate all other copies, thus guaranteeing that only one valid copy exists. Subsequent writes are performed locally, and the cache is responsible for responding to bus requests for such dirty lines. On a bus request, a cache that has a dirty copy of the line writes back to memory and the requesting cache gets the line from the memory. The cache controller maintains identical dual directory, one for bus accesses and the other for processor accesses. Since there is no invalidation line on the Multibus, the first write substitutes the invalidation signal. This feature causes some overhead due to memory updates each time a line is write-missed \cite{AB86}.
The Berkeley protocol proposed by Katz et. al. [KEW*85] uses an invalidation line in the bus to inform peer caches when a write operation is performed on a line. It reduces invalidation rate by distinguishing exclusive ownership from shared ownership and invalidating only when a shared line is modified. Instead of the dual directory scheme used in write-once, Berkeley scheme uses a dual-ported directory that allows concurrent reads and serializes reads and writes. This approach makes the implementation simpler. However, since the efficacy of cache memory originates from data locality in time and space, repeated invalidation and frequent line transfers are expected in schemes that use invalidation.

Instead of invalidation, an updated word is broadcast in the Dragon system [McC84] and caches with matching entries update the line. So, multiple caches may write to the same line and the last writer is responsible for updating the memory when the line is replaced. Bus traffic is lower than most other published protocols because a line transfer with an invalidation signal is replaced with a word transfer. So, the bigger the ratio of line size to a word, the better this scheme is than the write-once protocol. But contention for access to the cache directory could be significantly higher if there are frequent writes in the system.

The protocols considered thus far treat cache coherence problem divorced from synchronization. Bitar and Despain [BD86] propose a scheme in which the cache entertains lock and unlock commands in addition to read and write from the processor. On receiving a lock request, the cache broadcasts a write on the bus. If the write is allowed, the cache sets the state to locked and allows the processor to use the line. Otherwise, the line is in use by another cache. So, the requesting cache stores the address of the line in a special register called busy-wait register and the lock holder cache sets the state to lock-wait meaning there is at least one waiting processor. Upon receiving an unlock request, the cache changes the state to invalid and broadcasts unlock if its state was lock-wait. This lock scheme combines lock-based synchronization with the line transfer, thus performing locking in zero time. Read and write requests are processed like the Berkeley protocol. The non-identical dual directory can be used here since the dirty state is updated by the processor and the waiter state is updated by the bus. When there are more than one processors waiting for the same lock, all of them get the same high priority for bus access. When the lock is subsequently released, one
of the waiting processors (not necessarily the first one to make the request) obtains the lock. Thus there is room for starvation in this protocol. Furthermore, the scheme does not distinguish between lock requests for reading data from writing. Since it is envisaged that a large portion of shared data access is for reading and not writing, this scheme limits the potential concurrency.

3 A Lock-Based Protocol

We propose a lock-based protocol for consistency maintenance. The requests and responses exchanged between the processor, the cache, and the bus are illustrated in Figure 1. The processor and the cache together form a node of the shared memory multiprocessor. Each node is assigned a unique id which we will refer to in this paper as node-id. The handshake between the cache and the bus is explained in the following subsection. The cache entertains six requests from the processor: read, write, read-lock, write-lock, read-unlock, and write-unlock. Read and write are deemed as accesses to non-shared data and the cache processes them as would a uniprocessor cache. The granularity of a lock is a cache line. The processor waits until the request is satisfied by the cache.

Contentions for shared data engender a waiting queue such as the one shown in Figure 2. The
Figure 2: Example of a waiting queue

sequence of lock requests is P1:read-lock, P2:read-lock, P3:write-lock, P4:read-lock, P5:read-lock, and P6:write-lock. The first requester (P1) obtains a read-lock, and the following requester (P2) shares the lock since the lock type is read. P3 waits for the lock because its lock type is write. P4 and P5 wait after P3 to ensure fairness, even though current lock held by P1 and P2 is sharable. A peer-group is a group of read-lock requesters who concurrently share a lock ({P1,P2} and {P4, P5} are peer-groups). Similar to the early protocols that employ distributed cache directories, it is desirable to store the directory information in the cache line itself. To implement a queue, each directory entry of a cache line has a next-node field containing the node-id of the next waiting cache if any. When a lock is released, the cache sends a wake signal to the next waiting cache (if any). So caches with waiting states should monitor the signals on the bus for the line address and node-id. We first describe the cache controller actions considering only a single process per processor generating lock requests. We then extend the protocol to the case where we allow multiple processes per processor.

3.1 One process per processor

The possible states of a cache line are summarized in Table 1, where R,W are used to specify the lock type, T to signify the tail of the queue, V to indicate waiting state, and O for the ownership. State transitions are triggered by processor requests and/or bus activities. Note that the cache controllers only respond to lock and unlock requests on the bus since simple reads and writes are deemed to be for private lines. Therefore, the states in Table 1 apply only for shared lines obtained through lock requests. In the discussion to follow, we use lock and line interchangeably since lock acquisition is merged with the cache line transfer.
<table>
<thead>
<tr>
<th>States</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALID</td>
<td>The line is invalid</td>
</tr>
<tr>
<td>WO</td>
<td>Write lock owner.</td>
</tr>
<tr>
<td>WOT</td>
<td>Write lock owner at the tail.</td>
</tr>
<tr>
<td>WOV</td>
<td>Waiting for a write lock.</td>
</tr>
<tr>
<td>WOV T</td>
<td>Waiting for a write lock at the tail.</td>
</tr>
<tr>
<td>R</td>
<td>Read lock holder.</td>
</tr>
<tr>
<td>RV</td>
<td>Waiting for a read lock.</td>
</tr>
<tr>
<td>RO</td>
<td>Read lock owner.</td>
</tr>
<tr>
<td>ROT</td>
<td>Read lock owner at the tail.</td>
</tr>
<tr>
<td>ROV</td>
<td>Waiting for a read lock ownership.</td>
</tr>
<tr>
<td>ROVT</td>
<td>Waiting for a read lock ownership at the tail</td>
</tr>
<tr>
<td>O</td>
<td>Unlocked, but still an owner.</td>
</tr>
<tr>
<td>OT</td>
<td>Unlocked, but still an owner at the tail.</td>
</tr>
</tbody>
</table>

Table 1: States

An owner cache has the latest copy of the line, so it provides the line to the other caches when requested. The line is written back to memory when a write-lock owner releases the lock. There is at most one owner of a lock even when the lock is shared. A lock state with a T suffix denotes that the cache is at the tail of the waiting queue and should respond to subsequent requests for that lock. Only the first requester within a peer-group can be a tail or an owner. A shared lock is released when the size of the peer-group reaches zero, so caches with read-lock ownership or awaiting ownership keep the size of the peer-group in a count field. The ownership persists even after the line is unlocked at the owner cache. Assigning ownership to the first requester may result in unnecessary entries in the cache because it goes counter to the LRU (Least Recently Used) replacement principle. However, the alternative choice of giving ownership to the last one in a peer-group could generate more bus traffic to transfer the count variable to the new owner. The width of the count field is determined by the number of nodes in the system. Each cache line has
a directory entry (tag) with the fields as shown in Figure 3.

Figure 4 illustrates the state transitions. When a processor requests a read-lock, the cache broadcasts it on the bus resulting in one of following responses:

- A hit(M) signal is received on the bus. It means that the line is not locked by any cache. The memory system provides the line, and the cache changes the state to ROT since it is the first requester.

- A hit signal is received on the bus. The current lock owner sends this signal along with the cache line allowing the requester to share the lock. The receiving cache changes its state to R.

- A wait signal is received on the bus. A peer cache in state ROVT sends this signal with its own node-id. Since the wake signal (to be discussed shortly) is addressed to the first requester in a peer-group, this node-id is necessary for the waiting nodes to receive the signal correctly. The receiving cache stores this node-id in the next-node field, and changes its state to RV.

- A wait(T) signal is received on the bus. The signal comes from a cache in state WOVT or WOT, and signifies that the tail state is transferred to the requester. Therefore, the receiving cache changes its state to ROVT.

When a cache receives read-unlock from the processor, the state of the line is one of R, ROT, or RO. A cache line in the R state is simply changed to the state INVALID, and a read-unlock signal is broadcast on the bus to inform the owner to decrease the count. If the state is ROT or RO, it is changed to OT or O respectively after decrementing the count. The cache is still the owner even after its own processor releases the lock and is responsible for sending a wake signal when the count goes to zero. Even though we assume a single process per processor, a processor may request
Figure 4: State Transitions (Single Process per Processor)
a lock after releasing a lock, i.e., it may request a lock when the state of the line is OT or O. This case is not shown in Figure 4 since it is treated as a sub-case of multiple processes per processor.

In case of a write lock, it is not necessary for the owner to keep the count since only one writer is allowed at a time. If a wait(T) signal is received after broadcasting a write-lock, the state is changed from INVALID to WOVT. However, it ceases to be at the tail when any subsequent request for a lock is observed on the bus. On receiving an appropriate wake signal, the cache controller changes waiting states, WOV, WOVT, to owner states, WOT, WO respectively, and allows the processor to use the line. It is not necessary to broadcast a write-unlock. On receiving write-unlock request from the processor, the cache changes the state of the line to INVALID, sends wake signal enclosing the cache line to the next requester (if any) as indicated by the next-node field, and writes the line back to memory.

Signals broadcast on the bus are read-lock, write-lock, read-unlock, hit, hit(M), wait, wait(T), and wake. The wait signal is sent from the tail cache to the lock requester when the lock is unavailable at the moment, and wait(T) signal is sent when in addition the tail state is transferred to the requester. The wake signal is sent to notify that the lock is released to a cache whose node-id was stored in the next-node field of the tag entry for the line. The actions performed by a cache in the given state of a cache line on receipt of broadcast signals are summarized below:

Broadcast signal : read-lock

| ROT,OT: | increment(count); send(line); |
| ROVT: | increment(count); send(wait, my node-id); |
| WOVT, WOT: | state := WOV or WO; {respectively}. next-node := requester's node-id; send(wait(T), my node-id); |

Broadcast signal : write-lock

{prefix}T: state := {prefix};
next-node := requester's node-id ;
send(wait(T), my node-id) ;

Broadcast signal: read-unlock

\[
\begin{align*}
\text{ROT,RO:} & \quad \text{decrement(count)} ; \\
\text{OT,O:} & \quad \text{decrement(count)} ; \\
\text{If count} = 0, \text{then state} := \text{INVALID} ;
\end{align*}
\]

For simple reads and writes from the processor, we need two additional states VALID, and DIRTY to indicate the validity of the cache line and whether the line is to be written back on replacement. For the purposes of the lock requests, the actions of the cache controller are exactly the same from these two states as from the INVALID state. For replacement, the cache controller chooses a line whose state is either VALID, INVALID, or DIRTY. The replaced line is written back if it is DIRTY.

### 3.2 Multiple processes per processor

In this subsection, we extend the protocol of the earlier subsection to allow multiple processes per processor. If multiprocessing is allowed for each processor, there may be more than one process per processor waiting for the same lock. Therefore, each processor needs to maintain a local queue of processes for every lock unit. The state of the first requester for a given lock is stored in the corresponding line of a cache, and all the requesters including the first one are kept in the local queue. The operating system maintains the local queue, and all the queue elements have information about its own state and a pointer to the next waiting process (if any). When a lock is released, the cache controller has to determine the next state for the cache line. This determination has to be done based on the state of the next requester queued in the local queue. To facilitate this determination without holding the bus for arbitrary duration, we add a next-state field to each cache tag entry. Each cache line has a tag with the fields as shown in Figure 5.

The node with a tail process is responsible for responding to new requests for the same lock on the bus. Even though the first process of a local queue (the state stored in the cache controller)
<table>
<thead>
<tr>
<th>state</th>
<th>next-state</th>
<th>next-node</th>
<th>count</th>
</tr>
</thead>
</table>

Figure 5: Tag fields for multiple processes per processor

may not be the tail of the queue, there may be a request enqueued in the local queue that makes this node a tail node. Therefore, the state stored in the cache line must reflect whether this node is the tail node (i.e., there is a tail process in the local queue). We add suffixes $t'$, $t^w$ to the state of a cache line to indicate that the local queue has a tail process of request type read, and write respectively.

The additional states and the additional transitions required to incorporate multiprocessing are illustrated in Figure 6, where $*$ denotes one of R, RO, or WO. The state transitions in Figure 4 still apply. We describe below just the additional transitions shown in Figure 6. If the response to a read-lock request is wait(T), the operating system enqueues the requesting process in state ROVT in its local queue. The state of the cache line is $*t'$. If the response is wait, the process is enqueued in the local queue in state RV and the state of the cache line does not change. Once a cache line enters $*t'$ state, it does not receive any further local read-lock requests since the operating system can discern from its local queues that the requesting processes have to be simply enqueued in state RV. A write-lock request results in a cache line state change with a suffix $t^w$. Therefore, subsequent write-lock requests are processed locally by the operating system.

On an unlock request from a process, the operating system dequeues it from the head of the local queue. The next-state field in the cache line is the state of the process which is the new head of the local queue. The operating system generates an unlock request to the cache controller enclosing the state of the second requester (if any) in the local queue. Upon receiving an unlock request, the state of the cache line changes to the value stored in the next-state field and the next-state field changes to the enclosed request (if any) from the operating system. In updating the state, the $t'$, $t^w$ suffixes of the previous state is preserved except when the new state has T suffix which overrides $t'$, $t^w$. For example, if the current state of a line has $t'$ suffix, and the next-state is ROV, the new state on unlock request is ROVT$'$, but if the next-state is WOV'T, the new state is WOV'T.
Figure 6: Additional States and Additional Transitions
(Multiple Processes per Processor)

12
With an appropriate wake signal, the states \( *Vr \), \( *Vtw \) transit to \( *tr \), \( *tw \) similar to the transitions from \( *V \) to \( * \) in Figure 4. A cache line in a state with T suffix changes the suffix to \( tr \) or \( tw \) on receiving a new lock request for read and write, respectively. The exception to this rule is when the current state is ROT and the lock request is for read. In that case, the state remains unchanged, count is incremented by one, with the requester sharing the cache line.

A cache controller responds to bus requests for cache lines in states with \( tr \) or \( tw \) suffixes similar to the states with T suffixes in Section 3.1. However, storing the node-id of the next requester, maintaining count, and informing the operating system that a cache line has become available necessitate additional registers in the cache controller and we discuss that in the next section.

4 Implementation

4.1 Bus

None of the existing buses are adequate to implement our protocol. However, with minor modification, it is possible to implement our protocol in most current computer buses. We illustrate this fact by suggesting modification to the IEEE Futurebus [Bor87]. Futurebus is selected for the following reasons:

- It is a proposed IEEE standard for 32-bit computer buses.
- It supports asynchronous bus operations allowing caches to perform the state-dependent processing without worrying about bus-timeouts.
- It separates address cycle from data transfer allowing efficient snooping actions to be performed.
- It allows establishment of a virtual connection between a master and slaves that can be terminated at master's request.

On Futurebus, a bus master winning the bus arbitration places the address on AD (32 bits) and a command on CM (5 bits), and asserts an address strobe. The addressed slave places its status on ST (3 bits), and asserts an acknowledge line to establish a virtual connection. Four commands,
each on a separate CM line, are supported by Futurebus: read/write, bus-lock, block transfer, and broadcast; six encoded states are defined on ST lines. In our scheme, in addition to these commands, lock, unlock, and wake commands are required for a master, and four more states for the status of a slave, wait, wait(T), hit, and hit(M). Therefore, three more bits are needed for CM and one more bit for ST.

The bus handshake between nodes is illustrated in Figure 7. Other signals, such as address/data strobe, acknowledgement, and acknowledgement inverse, are omitted since they are irrelevant for the purposes of this paper. The bus transactions for a lock request are shown in Figure 7(a) and 7(b). The requesting node (master) places the line address on AD, and the lock command on CM. The responder (slave) is either the memory or the node which is at the tail of the queue for this line. The subsequent exchanges between the master and the slave are determined by the response on ST, and are summarized as follows:

- The response (Figure 7(a)) is hit or hit(M) - the slave sends the line to the master
The response (Figure 7(b)) is wait(T) - the master sends its node-id to the slave.

The response is wait - the master terminates the transaction.

As shown in Figure 7(c), a non-owner node releasing a read-lock broadcasts the line address on AD lines with the read-unlock command on CM lines. The current owner of the read-lock is the slave for this transaction. A wake operation (Figure 7(d)) needs to specify the id of the node to be waken up. This node-id is issued by the master following the line address, and then the line transfer is performed from the current owner to the waiting nodes. For these two transactions, the slave status is not germane to the cache protocol, but it is used by Futurebus to indicate the success or failure of the transaction.

4.2 Cache Controller

As described in Section 3, the actions of the cache controllers are triggered by either processor requests or bus activities, and these two kinds of actions contend for access to the cache directory. So, an appropriate interlock scheme in each controller is necessary to serialize these actions. Dual-ported read and the interlock scheme proposed in the Reference [KEW*85] are adequate for the directory as well as the cache data memory since most bus operations on shared data are reads, and there is neither invalidation nor multiple writers in our protocol. Non-identical dual directory [JSV88] is not feasible since the states of our protocol cannot be divided.

Each cache controller has two registers with the fields as shown in Figure 8. The buffer register records a bus request for a line for which the tail process is in the local queue; the wake-up register records the availability of a line for which there is a waiting process in the local queue. As we mentioned in Section 3.2, these events have to be conveyed to the operating system for appropriate action.
When the state of a line has either \( t' \) or \( t^w \) suffix, a bus request for the line results in the buffer register getting filled with the following information:

- **r/w** - Request type (read or write)
- **next-node** - Node-id of the cache controller initiating this request
- **line address** - Address of the requested cache line
- **v** - Validity of this register

When a wake signal is received on the bus, the cache controller updates the state of the cache line, enters the line address in the wake-up register, and sets the valid bit. Every time the operating system generates lock/unlock requests, it checks these two registers. If the valid bit is set, then the operating system updates the appropriate local queue, possibly involving scheduling decisions, before generating further local requests. Reading these registers clears the valid bits. Note that these are only one pair of registers in each cache controller. Therefore, a cache controller has to abort subsequent bus requests for lines in state with \( t' \), \( t^w \) suffixes until the operating system clears buffer register. Similarly, after one wake signal has been received, the cache controller has to abort subsequent wake signals until the operating system clears the wake-up register. However, we expect these situations to be very obscure and hence not to pose any serious limitation on system performance.

### 4.3 Cache Directory

A line is the unit of transfer between caches or between memory and a cache, and is also the granularity of locks in our protocol. Each directory entry in the cache memory has the followings fields in addition to the data line:

- **address tag** - The partial address of the line needed for associative look-up is stored here.
- **state** - The current state of the line is stored here. Five bits are needed to represent the 29 states of a cache line.
- **next-node** - The node-id of the next waiting processor is stored here.
• **count** - The number of processes that form a peer-group is kept here.

• **next-state** - The state of the second waiting process in the local queue is stored here.

When a cache line is in RV state, the cache controller should catch the wake signal addressed to the cache awaiting ownership for this line in the same peer-group. Recall that a cache awaiting read ownership for a line responds with its own node-id for subsequent read requests for that line. The requesters store this node-id in their next-node field. On receiving a matching wake signal on the bus, if the state of the line is RV, then the cache controller matches the next-node field against the node-id broadcast on the bus in the following cycle.

5 Summary and Conclusions

In this paper, we have presented a lock-based cache coherency protocol to merge synchronization and the actual data transfer. In this protocol, all the processes are required to use a read or a write lock to access shared data. Note that this requirement is very naturally satisfied given that processes request access to shared data through some mutual exclusion mechanism. For each lock, a queue of waiting processes is maintained. This queue ensures fairness to all the waiting processes, while avoiding unsuccessful retries from waiting nodes. The implementation of the queue is hardware-supported and is distributed over all the waiting nodes. The cache controller is slightly more complex than previous protocols requiring 15 states for a single process per processor and 29 states for multiple processes per processor.

We expect a significant reduction in bus traffic due to (a) merging synchronization with the data transfer, and (b) avoiding frequent line transfers and invalidations. We are currently in the process of simulating our scheme and obtaining quantitative measures of bus traffic reduction. With the reduced bus traffic, we expect the system to be more easily scalable to a large number of processors.

References


An Evaluation of Memory Management Structures for Object-based Systems*

Umakishore Ramachandran  M. Yousef Amin Khalidi

Technical Report GIT-ICS-88/53
December 1988

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Ga 30332-0280 USA

Abstract

Object-based operating systems have the desirable property of separating policies from mechanisms, while providing a protected procedure call interface for accessing system services. However, the kernel mechanisms in such systems rely very heavily on efficient memory management. A set of criteria for supporting the kernel mechanisms in object-based systems is presented. Six commercial memory management units (MMUs) are surveyed with respect to these criteria. We then present the design of a simple MMU tailored for object-based systems. The proposed design is an engineering solution combining the features available in commercial MMUs.

---

*This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.
1 Introduction

Traditional systems such as Unix, provide a protected procedure call interface for system services. However, in such systems the kernel tends to be monolithic and bulky since the policies and the mechanisms are all embedded in the kernel. In recent times, especially with the advent of local area networks, message-based operating systems have become popular [CZ83,ABB*86,RR81, HMSC87]. Policies are encoded in server processes and the kernel is lean and supports simply a message passing mechanism between processes. System services are requested by sending and receiving messages. However, the major drawback in such systems is the fact that a round-trip message is involved for requesting system services which can be an order of magnitude more expensive than a protected procedure call [Ram86].

Object-based systems [BHK*88,ABLN85,LS82,Nor87] view the resources of the system as a collection of objects. An object is the encapsulation of the code and data needed to implement the entry points in the object. These entry points provide the procedural interface for an activity to execute the code in the object. System services are requested via invocations (similar to a protected procedure call) on these entry points. At the same time the kernel is lean since it provides simply the invocation mechanism with the policies encoded in system objects outside the kernel. Thus object-based systems, at least in principle, combine the advantages of traditional and message-based systems.

However, implementation of object-based systems on conventional machine architectures can result in performance that is worse than message-based systems [RK89]. The focus of this paper is to identify the criteria for the efficient implementation of object-based operating systems (§3). Such systems are critically dependent on efficient memory management (§2). A qualitative evaluation is presented of commercial memory management units (MMUs) with respect to these criteria (§4). An MMU design is presented that best fits the model of computation advocated by object-based systems (§5). Extensions to the basic design are discussed in §6, and concluding remarks are given in §7.
2 Background

*Clouds* is a distributed object-based operating system being developed at Georgia Tech [BHK*88]. The discussion in this paper is presented in the context of Clouds to make it more precise. However, our work is applicable to object-based systems in general.

Clouds is intended to provide a unified environment over distributed hardware. Location independence for data as well as processing, atomicity of distributed computation, and fault-tolerance are some of the research goals of Clouds. *Objects* and *threads* are the basic building blocks of Clouds. Objects are passive entities and specify a *distinct* and *disjoint* piece of the global virtual address space that spans the entire network. An object is the encapsulation of the *code* and *data* needed to implement the *entry points* in the object. Thus a Clouds object can be considered syntactically equivalent to an abstract data type in the programming language parlance. Access to entry points in the object are accomplished through a capability mechanism in software.

Threads are the only active entities in the system. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object. A thread enters an object by invoking an entry point in the object. It then executes the code in the entry point, and returns to the caller object. Binding the object invocations to the entry points in the object takes place at execution time. Figure 1 shows the model of computation in Clouds. A thread in the course of its computation traverses the virtual address spaces of the objects that it invokes.

*Ra* [BHK87,BHK*89] is an operating system kernel designed to be the nucleus of the Clouds operating system. It currently runs on the Sun-3 architecture. The Clouds operating system is built on top of Ra using a collection of user and system objects. System objects are trusted software modules that reside in system space and provide the critical services that users expect from the operating system such as process control and object invocation. They are used to extend the kernel and to provide the system interface to user objects [BHK*89].
2.1 Ra Primitives

Ra defines and manages three primitive abstractions: segment, virtual space, and isiba. Segments serve as containers of data and may be viewed as uninterpreted sequences of bytes. The contents of a segment may only be accessed when the segment is attached to a range of virtual addresses. Segments persist until explicitly destroyed. Each segment resides in a partition that is responsible for providing backing store for the segment. A partition is a system object that realizes, maintains, and manipulates segments.

Virtual spaces abstract the notion of an addressing domain. A Ra virtual space is described by a segment called the virtual space descriptor (VSD) that contains a collection of windows. Each window is a data structure that identifies the segment that backs a range of addresses in the virtual space (Figures 2 and 3). Windows also describe the protection characteristics of ranges of the virtual space such as read-only or read-write. A virtual space is composed (or built) by a sequence of attach operations, each of which defines one of the virtual space’s windows. The attach operation defines a one-to-one mapping between virtual space addresses and segment locations such that virtual addresses starting at the designated address are associated with segment locations starting at the offset and continuing for the length specified.

The mapped ranges in a virtual space may not overlap, but ranges in the virtual space may remain unmapped. Thus, a single virtual address may not resolve to two or more segment locations but virtual spaces may have “holes” in them. Virtual spaces may share segments or ranges of locations in segments. Also, a segment may be mapped more than once to a single virtual space; that is, two or more windows in a single virtual space may refer to a single segment. Multiple mapping can happen in two ways: either two or more windows in the space refer to disjoint regions of a single segment, or two or more windows in the space refer to overlapping regions of a single segment. In the second case, distinct virtual addresses in the same space resolve to the same segment location.

Ra isibas are an abstraction of the fundamental notion of computation or activity and can be thought of as light-weight processes. Isibas may be used as daemons within the kernel or
they may be associated with a virtual space (called P space) to implement a user process. User processes are used to implement Clouds threads.

A Clouds object is a virtual space (called O space) that consists of code and data segments. The code segment of an O space has entry points that can be invoked by user processes.

The machine virtual address space consists of three distinct regions that are called K, O, and P hardware spaces for kernel, object, and process, respectively. The kernel is always mapped in the K hardware space. A Process is mapped (or installed) into the P hardware space and unmapped on context switch.

A process consists of an isiba and a virtual space (P space). The object in which a process is currently executing is mapped into the hardware O space. Object invocation involves mapping the required memory segments of the object into the address space of the invoking process by installing the object as the current O space with the process’s P space. On object call, the O space of the called object is installed, and on object return, the O space of the caller object is installed.

Ra advocates a model of programming with a large number of possibly small-sized segments. Current implementation of Ra on the Sun-3 allows around 100 windows.

3 Comparison Criteria

As can be seen from the description of Clouds and Ra, an efficient implementation relies very heavily on effective management of virtual memory. In general, most MMUs have the following features:

- An address space pointer that points to the address space mapping table, also referred to as the page table. Address space switching is usually accomplished with a hardware pointer update in most MMUs.

- A page table that holds mapping and protection information. In most MMUs the page tables are kept in physical memory. The page table may be organized as a linear table
(e.g. Vax-11), a tree (e.g. Motorola 68000 [Mot86]), or a hash table (e.g. IBM PC/RT [IBM86], HP Precision Architecture [MLM*86]). The latter two organizations are aimed at minimizing the amount of physical memory space that has to be reserved for holding the page tables when the virtual memory space is large and sparse. Organization as a linear table necessitates the use of other mechanisms, such as paging the page tables themselves, to limit the size of the page tables in physical memory.

- A Translation lookaside buffer (TLB) that holds recent virtual to physical address translations. A TLB is a high-speed associative cache and its purpose is to eliminate the need to access page tables that are often kept in main memory on each virtual-to-physical translation. In many MMUs, TLB entries are invalidated on every address space switch.

In particular, the following five MMU features are required to support an efficient implementation of Clouds/Ra:

C1. **The ability to cache TLB information across object invocation.** The cost of flushing TLB entries at object call/return is an implicit cost that affects mostly user mode time.

C2. **The ability to implement an object call/return by performing a small number of operations on the MMU.** The cost of required MMU operations is an explicit part of object invocation. This cost is quantified by the number of MMU registers and page table entries that need to be modified on each object call/return.

C3. **The ability to represent sparse address spaces efficiently.** Inefficient representation of sparse address spaces results in page tables that have large memory requirements and lengthy initialization time, relative to the size of address space actually allocated.

C4. **The ability to share memory among spaces.** This is required to support sharing of Ra segments among virtual spaces.

C5. **The ability to allocate/deallocate ranges of addresses easily.** The ability to allocate/deallocate ranges of addresses is related to sharing of memory segments among spaces, because ranges of addresses are allocated/deallocated at the segment level. The number of page table and
MMU operations needed to allocate/deallocate a range of addresses affects the cost of attaching and detaching segments.

4 Survey of Commercial MMUs

Several commercial MMUs are surveyed in this section. For each MMU, a description of the MMU and the virtual address translation process are given, an implementation of the Ra kernel virtual memory mechanisms using this MMU is sketched, and an evaluation of the MMU with respect to the criteria of §3 is presented. We concentrate on architectural features of the MMUs, rather than implementation restrictions.

4.1 HP Precision Architecture

4.1.1 MMU Overview

Virtual memory in the HP Precision Architecture [MLM*86] is built around one large virtual address space that is structured as a set of address spaces (segments) each containing $2^{32}$ bytes. There are $2^{16}$ or $2^{32}$ spaces (depending on the particular implementation), and each space is composed of 2K pages. Each address space has a space identifier specified in a space addressing register. A virtual address is the concatenation of a space identifier with a 32-bit displacement (Figure 4). There are 8 space addressing registers (SR 0 through SR 7) and each instruction that generates a virtual address (either explicitly or implicitly) selects one of these 8 registers. A virtual address is defined globally and has the same meaning when used by any process.

A TLB is used to cache recent virtual-to-physical address translations. Virtual to physical address mappings are maintained in a single physical page directory (PDIR). Because of the very large virtual address space, PDIR contains an entry per physical page and a hash table is used to locate a desired entry given a virtual address (this organization is also called an inverted page table). Virtual address translation proceeds as follows: Given a virtual address, the TLB is searched for the desired virtual-to-physical translation. On a TLB miss the PDIR is searched with the hashed virtual address. Virtual addresses that hash to the same hash table entry are
linked together. If the PDIR search fails, a page-fault is generated.

The use of an inverted page table prohibits two different virtual addresses from mapping into the same physical address (i.e. no aliasing). Sharing of memory between different processes is possible only through hardware space sharing.

4.1.2 Ra Implementation

Ra allows a large number of possibly small-sized segments. Any Ra segment is potentially sharable between several Ra spaces. Because sharing of memory in the HP architecture is possible only through hardware space sharing, a hardware space maps exactly one Ra segment.

Table 1 shows a possible assignment of the eight space registers to Ra spaces and segments. Note that SR 5 through SR 7 can be modified in privileged mode only.

The K space of Ra is contained in one hardware space that is permanently assigned to SR 7. One code segment of the currently invoked object is assigned to SR 6, and the stack segment of the current process is assigned to SR 5. SR 4 and SR 3 are changed dynamically to point to different segments that are attached to the current O and P spaces, respectively. In this implementation, the compiler is responsible for loading the hardware space registers SR 4 and SR 3 with identifiers to spaces containing the required segments. The space identifiers are set up by the kernel when the corresponding segments are activated. Therefore, the mapping of segment names known by the compiler (external names) to space identifiers (internal names) have to be communicated by the kernel to the compiler runtime system. SR 2 and SR 1 are used to point to spaces that contain the mappings from Ra segment names to the hardware space identifiers. The kernel is responsible for creating and updating these mappings, and the compiler is responsible for using the mappings to load SR 4 and SR 3. Note that the alternative of letting the kernel update SR 4 and SR 3 eliminates the need for the external-to-internal mapping segments. However, in this case, special kernel calls are required to set up the space registers for segments that are not currently mapped.

Object invocation proceeds by changing SR 6, SR 4, and SR 2 to map a different O space.
On process context switch, SR 5, SR 3, and SR 1 are changed to map a different P space, and SR 6, SR 4 and SR 2 are changed to map a different O space, if need be. No TLB flushing is required.

4.1.3 Evaluation

The following points correspond to the five criteria in §3:

C1. TLB entries are cached across object invocation.

C2. Three MMU operations are needed per object call or return. Process context switch requires at most six MMU operations.

C3. The inverted page table organization lends itself to an efficient representation of sparse address spaces.

C4. User software is complicated by the need to load SR 3 and SR 4 dynamically before accessing a segment that is not already mapped by one of these two registers. The compiler-generated code has to manage the usage of SR 3 and SR 4, which could result in performance degradation if there are many segments to access [PGH*84], because of the frequent need to load SR 3 and SR 4.

C5. Only one MMU operation is needed to map a segment to a virtual space.

A potential problem with this implementation is the possibility of protection violation. Since the space registers are manipulated by non-kernel code the security of the system can be compromised by malicious code that loads these registers with unauthorized identifiers.

The architecture defines a protection mechanism based on access identifiers. Each page may belong to one of \(2^{15}\) protection groups. A process is allowed to have access to at most four protection groups at a time. A possible implementation of secure access to segments is to assign a protection identifier to each segment, and allow each process to have access to at most four segments at a time. When a process tries to access a segment that is not in one of the four
current protection groups a protection fault is generated. When the protection fault occurs the
kernel checks whether the process should be allowed to access the segment. If the process is
allowed to proceed the kernel adjusts the membership of the process in the protection groups
and resumes its execution.

4.2 IBM RT/PC

4.2.1 MMU Overview

The IBM RT/PC MMU [IBM86] organizes memory as one large virtual address space of size
$2^{40}$ bytes that consists of 4096 segments of size 256M bytes each. The CPU generates 32-bit
addresses, and each process has a 4G bytes address space that is divided into sixteen segments.
There are sixteen segment registers that define the virtual space of a process at any point of
time. The segment registers are each 12 bits long, and can only be changed in privileged mode.

The MMU maintains a TLB of recent virtual-to-physical address translations. Address
translation proceeds as follows (see Figure 5): The high order four bits of the 32-bit CPU
address are used to index into one of the sixteen segment registers. The contents of the selected
segment register (12 bits) are concatenated with the lower 28 bits of the CPU address. The
resulting 40-bit address is then translated into a physical address by first looking up the TLB,
and on a TLB miss searching an inverted page table. As with other schemes that utilize an
inverted page table, address aliasing is not allowed and sharing is done at the hardware segment
level.

4.2.2 Ra Implementation

Similar to the HP Precision Architecture, IBM RT/PC allows sharing only at the hardware
segment level, and a hardware segment maps exactly one Ra segment. Unlike the HP system,
however, each process sees a flat 4G bytes address space. Therefore, the available 16 segment
registers have to be divided among the K, P, and O spaces.
A possible allocation of the 16 segment registers uses SR 15 to map the kernel, SR 1 through SR 6 to map the current O space, and SR 7 through SR 14 to map the current P space. (SR 0 is reserved for use by the kernel.) With this organization, a Ra virtual space (O or P) consists of at most 7 attached segments at a time.

On an object call/return SR1-SR6 are changed, and on a process switch SR7 through SR14 and SR1 through SR6 are changed. Flushing the TLB is not required.

4.2.3 Evaluation

The following points correspond to the five criteria in §3:

C1. TLB entries are cached across object invocation.

C2. Seven MMU operations are needed per object call or return. Process context switch requires at most fourteen MMU operations.

C3. The inverted page table organization lends itself to an efficient representation of sparse address spaces.

C4. The ability to share segments is restricted to seven segments. More importantly, an O or P virtual space may have at most seven segments attached to it at a time. This restriction limits the model of computation advocated by Clouds with a large number of possibly small-sized segments. The compiler could generate code to explicitly attach and detach segments to access more than 7 segments. However, this forces the programmer to a style of programming where he is aware of the small number of attached segments, and the need to manage attaching/detaching them. In addition, the overhead of frequent kernel intervention to attach/detach segments may be large.

C5. Only one MMU operation is needed to map a segment to a virtual space.
4.3 Sun-3 MMU

4.3.1 MMU Overview

The Sun-3 MMU [Sun86] provides each process with a virtual address space of size $2^{28}$ bytes. The 28-bit virtual address is divided as shown in Figure 6 into an 11-bit segment number, a 4-bit page map entry number, and a 13-bit page offset (each page is 8K bytes long). The MMU contains 8 segment maps, 256 page map entry groups (PMEGs), and a 3-bit context register. Each PMEG contains sixteen page map entries, each of which maps one physical page.

Address translation proceeds as follows: One of the 8 segment maps is chosen depending on the current value of the context register. The selected segment map is indexed by the 11-bit segment number from the virtual address. Each segment map entry is 8 bits long, and selects one of the 256 PMEGs. The selected PMEG is indexed by the 4-bit page map entry number from the virtual address to select one of sixteen page map entries in each group. The selected page map entry contains the physical page number (plus protection information).

There is no TLB in the Sun-3 MMU. Instead, the segment maps and PMEGs are contained in the MMU itself, and serve as a cache of the mapping information maintained in main memory by the kernel. The kernel is responsible for setting up the segment maps, the PMEGs, and the context register. When address translation results in an invalid page map entry a fault is generated and the software is responsible for servicing the fault.

4.3.2 Ra Implementation

The current Ra implementation [BHK*89,BK88] runs on the Sun-3 architecture, and this subsection describes this implementation.

Each context is divided into three regions corresponding to the O, P, and K spaces. The K space is permanently mapped into all contexts. One of the 256 PMEGs is reserved and its 16 page map entries are initialized to “page invalid”. Any segment map entry that does not map a valid range of addresses contains the index of the invalid PMEG.
To install an O (or a P) space into a particular context, the segment map entries that map the corresponding address range of the space has to be initialized. The kernel initializes all such entries to the invalid PMEG, and may also initialize some of the entries to valid PMEGs. If some segment map entry is left uninitialized the process running in this context will access data belonging to the object/process that last used the same context.

Object invocation and process context switching usually involve choosing a context to use, and installing an O and/or a P space in the selected context. The kernel maintains an 8-element array that keeps track of the currently installed O and P spaces in each context. Process context switching is handled as follows: If there exists a context that has the desired P and O spaces already installed a change to that context is made (by writing the number of the selected context in the context register), and no other MMU operations are necessary. Otherwise, if there is a context that has the required P space already installed a change of context is made and the required O space is installed. Otherwise, a change of context is made to the least recently used context, and the desired O and P spaces are installed. Object invocation is similar, and is handled as follows: If the required O space is already installed in a context with the same P space (or with a null P space), a switch to that context is made. Otherwise, if there is a free context (null O and P spaces), that context is used. Otherwise, the O space is installed in the current context.

Object invocation requires mapping an O space with the current P space. Object invocation is similar to process switching in that the use of a different context may be desirable in hope of caching O space mappings in the MMU across object invocations. For example, two contexts may have two different O spaces and the same P space installed. If the two objects represented by the O spaces exhibit invocation locality, subsequent object invocations require only a change of context (see next subsection).

4.3.3 Evaluation

The following points correspond to the five criteria in §3:
C1. The Sun-3 MMU does not use a TLB. Instead, the 8 segment maps and PMEGs serve as a cache of recently used mappings. Though it is easy to see how to use the 8 contexts to cache process mappings in a conventional system, it is not clear how best to utilize these maps to cache mapping information across object invocations.

C2. Installing an O space may require a large number of MMU operations. In the current Ra implementation almost 650 segment map entries are needed to fully map an O space. Depending on the size of the O space previously installed in the selected context a large number of segment map entries may need to be initialized.

C3. The segment map is a linear table and requires initializing all entries to map a space. Similar remarks apply as in C2 above.

C4. By assigning the same page number in different PMEG entries, sharing of segments between spaces is possible.

C5. Allocating a range of addresses need not be reflected in the MMU registers immediately. Instead, when the currently executing process faults on an invalid PMEG, the kernel can set up part of the segment mappings in the MMU. Deallocating a range of addresses require traversing the segment map, modifying entries to point to the invalid PMEG, and possibly invalidating several PMEGs.

Caching of translation information is achieved through the use of the eight contexts. In a conventional system, the use of contexts to cache mappings is straightforward: A process occupies a whole context. If a process is already mapped in one of the eight contexts, a change to that context is all that is needed to effect a process switch. Otherwise, the least-recently used context is reused and the process is mapped into that context.

In an object-based system, however, it is not clear how best to make use of the eight contexts to cache mappings across object invocation and process context switching. Unlike conventional systems, in object-based systems the O and P spaces are handled separately. As explained before, a decision on which context to use is made at each object invocation and context switch.
The use of the eight contexts has to be scheduled. Since the kernel cannot predict the order in which objects are invoked optimal scheduling is not possible. The algorithm used in choosing a context by Ra on process switches and object invocation (§4.3.2) is only a heuristic, and other variations on the algorithm are possible.

4.4 Vax-11

4.4.1 MMU Overview

The Vax architecture [Dig86] provides each process with a 4G bytes paged virtual memory with the system kernel mapped into one half of the memory space (system space), and the currently executing process mapped into the other half (process space). The process space is further divided into P0 and P1 spaces. Program images and most of their data reside in P0 space, while P1 space contains the process stacks and other data. A page table that resides in physical memory describes the system space, and two separate page tables describe the currently mapped process space. The process page tables reside in system virtual memory. A pair of system registers (P0BR and P0LR) describe the starting address and length of P0 page table. The P1 page table is described in a similar manner using P1BR and P1LR registers. The location and size of the system page table in physical memory are specified using the SBR and SLR, respectively.

The hardware provides a TLB for caching recent virtual-to-physical memory translations. The TLB is split into two halves: one half caches translations from the system space, while the other half caches translations from the process space. The latter part is invalidated on each process context switch.

The high order two bits of the 32-bit virtual address are used to select one of P0, P1, or system space page tables. The remaining 30 bits are split into a 21-bit virtual page number (VPN) and a 9-bit page offset (each page is 512 bytes long). Address translation proceeds by searching the TLB for translation of the VPN. If no such translation is found, the VPN is used to index the page table selected by the high order two bits of the virtual address. The page table entry (PTE) found by indexing the page table contains the physical page number, or indicates
a missing page and a fault is generated. Note that because process page tables reside in virtual memory, a search of the system page table may be necessary to locate the translation of the system space page that contains the desired PTE. This implies that as part of handling a process TLB miss, referencing the process PTE may cause another TLB miss.

4.4.2 Ra Implementation

The first implementation of Clouds [Spa86] was on the Vax architecture, and the following description of a possible implementation of Ra resembles the first implementation.

The virtual address space of the processor is divided as follows: The kernel is permanently mapped into the system space. The currently invoked object is mapped into the P0 space, and is represented by a page table. Similarly, the current P space is mapped into the P1 space and is also represented by a page table. Sharing of memory ranges is done by having page table entries from different page tables point to the same shared physical pages (the Vax architecture allows aliasing). A modification to a PTE is propagated by the kernel to other PTEs sharing the same physical page.

Object invocation requires changing the P0BR and P0LR registers, and flushing the TLB. A process context switch changes P1BR and P1LR to map a new P space, changes P0BR and P0LR to map a new O space (if need be), and flushes the TLB.

4.4.3 Evaluation

The following points correspond to the five criteria in §3:

C1. The portion of the TLB caching user address translations must be flushed at object call/return.

C2. Only two MMU operations (other than flushing the TLB) are needed to implement object invocation.
C3. The linear page tables do not represent sparse address spaces efficiently. A large sparse address space requires initialization time and memory requirements on the order of the highest address used and not the actual amount of memory allocated. One way to address this problem is to exploit the fact that the page tables describing user space reside in system space virtual memory. By manipulating the manner in which the page tables themselves are mapped into system virtual space it is possible to represent sparse spaces more efficiently [KB84] (at the expense of a complex system software). However, attaching/detaching segments to an address range requires a linear traversal on the order of the address range size (C5).

C4. Sharing of segments among spaces is possible and is implemented by having page table entries from different page tables point to the same shared physical pages. The kernel is required to update all page tables sharing a segment whenever the segment (or a part thereof) is paged-in/paged-out.

C5. Deallocating/allocating an address range in a virtual space requires traversing the page table describing the space and changing each PTE in the range. Additionally, when deallocating address ranges the affected PTEs have to be flushed from the TLB.

4.5 MC68851 MMU

4.5.1 MMU Overview

The MC68851 MMU [Mot86] provides each process with a $2^{32}$ bytes virtual space. One half of the virtual space is system space, while the other half is process space. Each space is represented by a page table in physical memory. The base address of the two page tables are contained in MMU registers.\footnote{A third page table is used to map DMA devices.} The page tables are organized as trees. The depth of the tree as well as the page size can be selected by the software. The software can arrange the page table as a linear table or as a tree with up to four levels.\footnote{Using the MC68020 function code signals, the tree can have up to five levels.}
Figure 7 shows how a virtual address is used to index the page table. The index fields TIA, TIB, TIC, and TID specify the number of bits of the virtual address to be used at each level of the translation tree. The IS field is used to set the maximum process virtual space size to $2^{32-\text{IS}}$. The page size is set to $2^{Ps}$. An MMU control register is initialized by the software to contain the lengths of the fields IS, TIA, TIB, TIC, TID, and PS. Note that all fields except TIA and PS can be zero, and that the sum of TIA + TIB + TIC + TID + PS + IS must be equal to 32 (each virtual address is 32 bits).

A TLB is used to cache recent virtual-to-physical address translations. On a context switch, the kernel writes the base address of the new user page table into an MMU register. The MMU keeps track of the last eight recently-used base addresses in a table, and uses the index into the table as a process identifier. Addresses from the process space are tagged with this identifier which is also stored as part of the tag in the TLB, thus eliminating the need for flushing the TLB on context switches.

An MMU register maintains the tag of the currently running process. Address translation proceeds as follows: The virtual page number portion of the virtual address (specified by TIA, TIB, TIC and TID) plus the tag of the currently running process are used to search the TLB. If no match is found, the page-table is accessed as shown in Figure 7. Each sub-field of the virtual page number is used to index a different level of the page-table tree. When the desired page table entry is found, it is loaded into the TLB.

4.5.2 Ra Implementation

The MC68851 provides a separate page table for the system space that is used to map the K space. The process space is divided into two equal parts to map the P and O spaces. Because there is only one page table for the process space, the first-level of the tree (defined by the TIA field) is divided equally between the O and P spaces. The first level of the tree should not have many entries since the number of page table operations required to perform object invocation is equal to half the number of entries at this level. With regard to the total number of levels
in the page table, the more levels there are, the more flexibility the kernel has in storing and manipulating the page table. However, TLB miss handling time increases with the number of levels. Using TIA, TIB, and TIC (and setting TID to zero) produces page tables with three levels, which seems as an appropriate compromise. However, values of TIA, TIB, TIC, and PS have to be chosen, and a possible assignment of these fields is shown in Table 2.

Object invocation is implemented by changing the entries corresponding to the O space in the first-level page table of the currently executing process. Process context switching requires changing an MMU register to point to a new page table, and modifying the entries in the first level corresponding to the O space.

The TLB has to be flushed at object invocation and process context switches. The “process” tags kept in each TLB entry is sufficient to cache address translations across process context switches in a conventional system. However, under Ra, “process” addresses refer to O and P addresses, and therefore, the MC68551 tags are not sufficient to cache TLB translations across object invocations and context switches. If the TLB is not flushed across object invocations, the process will continue to access the previous O space (instead of the current O space) through the TLB entries thus compromising correctness and the data encapsulation property of the object-based system. On the other hand, if the TLB is not flushed across process context switches, there is a possibility for duplicate TLB entries for the same O space with different process tags. While this duplication does not compromise system security, it could lead to inconsistencies if the mapping information for an O space that is multiply cached changes.

4.5.3 Evaluation

The following points correspond to the five criteria in §3:

C1. TLB must be flushed across object invocation.

C2. Depending on the organization of the page table tree, only a few first level entries need to be modified to effect object invocation.
C3. The organization of the page table as a tree lends itself to a more efficient representation of sparse address spaces than a linear table.

C4. Sharing of memory among spaces is possible by mapping virtual addresses from different spaces to the same physical addresses. Different page table entries can map to the same physical addresses. Alternatively, different page tables can share common subtrees that represent shared segments.

C5. Allocating/deallocating ranges of addresses requires manipulating the page tables. The tree organization of the page table makes it possible to allocate/deallocate ranges of addresses by modifying a subtree of the page table. The number of required operations depends on the size of address range, starting position in the address space, and the structure of the tree.

4.6 SPUR Architecture

4.6.1 MMU Overview

The SPUR architecture [WEG87] supports a single, segmented global virtual address space that consists of 256 one gigabyte segments. A segment consists of 256K pages and the page size is 4K bytes. Each process has an address space of $2^{32}$ bytes that consists of four segments. The high order two bits of the process virtual address selects one of four Global Segment Registers (GSR) that are maintained by the kernel. Each GSR is eight bits long and specifies one of the 256 segments. The segment number contained in the selected GSR is concatenated with the virtual page number (VPN) and page offset to form a Global Virtual Address (GVA), as shown in Figure 8.

The SPUR architecture uses virtual caches to cache data and translation information. The cache associates virtual address tags with data blocks. SPUR uses an in-cache address translation algorithm [WEG*86] and does not use a separate TLB. A single three-level page table that resides in virtual memory maps the entire global virtual address space. The process of translating a GVA to a physical address uses the cache, and if necessary inspects the global page table.
Because the page table resides in virtual memory, and it maps the entire global virtual space, a portion of the page table maps itself. The third (highest) level of the page-table resides at a well-known physical memory location.

Since SPUR uses a virtual cache, memory accesses for data and instructions that are available in the cache do not require translating the GVA into a physical address. When there is a cache miss, the GVA is translated into a physical address by inspecting the page-table. Since the page-table resides in virtual memory the cache is searched for the (first-level) PTE that maps the page containing the required datum/instruction. If the PTE is not found, the cache is searched for the (second-level) PTE that maps the page containing the first-level PTE. If the second-level PTE is not found in the cache, the root (third-level) of the tree is accessed from a well-known physical memory address.

The only mechanism for sharing data among processes is by sharing hardware segments. If any portion of a segment is shared by two processes, the whole segment is shared. Virtual address aliasing is prohibited since the virtual cache cannot handle aliasing.

4.6.2 Ra Implementation

The SPUR architecture presents an interesting challenge for an efficient implementation of Ra. Though it is clear that one of the four GSRs should permanently select a shared hardware segment that maps the K space, it is not clear how to construct the O and P spaces out of the remaining three GSRs.

An implementation on the SPUR architecture uses one GSR to map-in one code segment from the O space, one GSR to map-in the stack segment from the P space, and one GSR to map-in any other O or P segment that needs to be accessed. The compiler has to generate code for explicitly attaching/detaching any segment (other than the stack and code segments). Such an implementation presents a model of programming where only one segment (other than the stack and code segments) is attached at a time, and forces kernel intervention each time any other segment needs to be accessed.
An alternative implementation represents each Ra virtual space as one of the 256 hardware segments, and uses software techniques to enable different virtual spaces to share segments. The four GSRs that define the current process virtual space are assigned as follows: the first GSR is used to map the current 0 space, the second GSR is used to map the current P space, and the third GSR is used to map the K space. The fourth GSR is not used. Since each Ra virtual space is mapped using a hardware segment a portion of the global virtual table maps each virtual space. Sharing of segments among virtual spaces is implemented by virtual address aliasing.

When the current O or P space has at least one segment that is shared in read-write mode with another space, the cache must be flushed on the next object invocation or context switch. Depending on the amount of sharing, this implementation can be very costly because flushing the cache removes cached address translation data (similar to TLB flushing) and cached data. In systems that use physically-addressed data caches, the data cache need not be flushed even if the TLB is flushed on each object invocation and context switch. In addition, the actual operation of flushing the cache can be very costly. The cache size is 128K bytes arranged as 32 byte blocks. Using the flush instruction, flushing the cache may require up to 4096 operations.

A modification to the above scheme does not flush the cache on object invocation or context switch, but instead flushes individual cache lines only when necessary. When a segment is attached to a virtual space VS1, all PTEs corresponding to the segment in VS1 are initialized to invalid. When a page-fault occurs on one of these PTEs, the kernel locates the required physical page. If the physical page is currently mapped as part of another virtual space VS2, the PTE corresponding to the page in VS2 is invalidated, the cache lines containing the PTE and data from the page are flushed, and the PTE corresponding to the page in VS1 is set to point to the physical page.

The choice of one of these two implementations depends on the amount of sharing among virtual spaces. If only a few shared pages are touched during an object invocation or a context switch, the latter approach is better. On the other hand, if the second approach generates more than 4096 cache flush operations per object invocation because of shared pages, the first approach is more efficient.
4.6.3 Evaluation

The following points correspond to the five criteria in §3:

C1. Cached address translation information and data of shared segments have to be flushed either on object invocation and process context switch, or during execution, depending on the implementation.

C2. Assuming that the cache is not flushed, an object call/return requires changing one GSR only. If the cache is flushed, up to 4096 additional operations may be necessary. The extra costs of faulting on shared data and flushing cache lines during execution increases execution time depending on the number of shared pages.

C3. The structure of the page table describing each virtual space is more efficient than a simple linear table. However, the number of entries that needs to be initialized at the second level of the page table to represent a segment is proportional to the size of the segment. Specifically, \( \lceil s/4K \rceil \) entries must be initialized for a segment of size \( s \) bytes.

C4. The scheme we presented allows the sharing of segments among virtual spaces. However, the cache must be flushed when sharing read-write segments.

C5. The number of page table entries that need to be modified to allocate/deallocate a segment is proportional to the size of the segment.

Another issue raised by the use of a virtual cache, is the need to flush cache entries when modifying page table entries. Since part of the PTE is included in each data block, all data blocks in the cache that belong to a page have to be flushed when the PTE is modified. In a system that uses a physically-addressed cache and a TLB, only one TLB entry may need to be flushed when modifying a page table entry. In Ra, page table entries are modified when allocating/deallocating ranges, and therefore, cache entries must be flushed when attaching/detaching segments.

As with other MMUs, the kernel has to maintain two sets of page tables: the hardware-imposed page tables, and the software page tables (VSD) that describe each virtual space. The
virtual subsystem of the kernel has to build and maintain the MMU page table information from
the VSDs, and has to keep the two sets of tables up to date. When there is a mismatch between
the MMU tables and the kernel tables, system software becomes complex and performance may
be adversely affected. In §5, we propose an MMU that solves this problem.

4.7 Other Systems

We have studied several other memory management architectures, including the ones used in
the IBM System/38 [IBM80], Intel 80386 [Int86], Motorola MC 68030 [HM87], and MIPS RISC
CPU [DMM86].

The IBM System/38 uses one large global address space, and an inverted page table. Though
the CPU architecture of IBM System/38 is capability-based, the memory management archi-
tecture is similar to the IBM PC/RT, and similar remarks apply to IBM System/38 as those
presented in §4.2.

The MC 68030 incorporates an on-chip MMU that is very similar to the MC 68851 MMU.
The MC 68030 MMU is a subset of the MC 68851, with a smaller TLB. The discussion of the
MC 68851 presented in §4.5 apply to the MC 68030 MMU.

The Intel 80386 has user-visible hardware segment registers (similar to the HP Precision
Architecture). Each process has associated with it a segment table and a page table. Each
address generated by the CPU references one of several segment registers. The contents of the
segment register is used to index into a segment table, and a “flat” virtual address is computed.
The page tables are then used to translate this flat address into a physical address. The Intel
80386 poses issues similar to those already covered in this section.

The MIPS RISC CPU implements a simple MMU tailored to the Unix operating system. The
MMU has a TLB that is tagged with process identifiers, and a simple software-based mechanism
is used for reloading the TLB on address translation misses. Our proposed MMU (§5) borrows
some ideas from the MIPS MMU.
5 Proposed MMU

5.1 Design Goals

We used the following goals as a starting point for the design of an MMU tailored to Ra:

- The MMU should meet the five criteria C1–C5 of §3.
- The MMU should be usable for conventional systems as well as object-based systems.
- The MMU should be easy to implement.
- The MMU should be easy to interface to existing CPUs with 32-bit addresses.

5.2 Overview

It is clear that the MMU should cache address translation information across object invocation (criterion C1), and our design achieves this goal by appropriately tagging virtual addresses. There is one important difference between our address tag and other MMUs that tag TLB entries: A distinction is made between O space addresses and P space addresses, and they are tagged differently. In addition, to save on the number of tag bits, addresses are tagged with object and process aliases (see §5.3).

As is shown in §4, the structure of the page tables affects C2–C5 considerably. MMUs usually dictate a page table format to enable the hardware to search the tables on TLB misses. Unless this page table format meets the OS requirements, a mismatch occurs that may adversely influence performance and increase software complexity. It is important to note that with a large and carefully designed TLB most of the translations come from the TLB, and the page tables are consulted only on TLB misses.

From our Ra implementation experience [BK88], we have found that the VSD structure described in §2 best suites the software implementation. The kernel maintains a VSD structure per (O and P) space. The window descriptors in the VSD describe mappings between address ranges in the virtual space and segments, and an inverted page table is used for each active
segment. Though it is possible to implement an MMU that knows about the structure of the VSD, we chose instead a RISC-like approach in the design of our MMU.

Our MMU does not know about page tables nor cares about the structure of the kernel page tables. Instead, it provides a large TLB, with appropriate tagging of virtual addresses, and the kernel is responsible for reloading the TLB on address translation misses. The structure of the kernel tables are not known to the MMU nor are they influenced by the MMU design. Instead, they are dictated by the kernel requirements. In case of the Ra kernel, each VS is described by a VSD, and each segment is described by a hash table. Ra maintains the VSDs, and traverses them on TLB misses.

5.3 Scheme

Figure 9 depicts the 32-bit CPU virtual address space. The O space resides in the first gigabyte, while the second gigabyte is reserved for the P space. The upper two gigabytes are reserved for the K space, and are divided into two sub-spaces, k_r and k_p. Addresses from the k_p sub-space are deemed as physical addresses and hence do not require translation. The MMU ignores all such addresses. Addresses from the k_r sub-space are kernel virtual addresses, and their translation is discussed shortly. Addresses from O and P spaces are tagged with object and process name aliases, respectively.

Figure 10 shows the structure of the MMU. The Object Name Register (ONR) and Process Name Register (PNR) hold the names (a bits long) of the current object and process, respectively. The kernel is responsible for setting these registers on object invocations and process context switches. The Alias Table (AT) is maintained by the MMU. The AT maintains the names of the last n used object and process names. The index of a space name into the AT is used as a name alias for addresses coming from this space. The Object Alias Register (OAR) and Process Alias Register (PAR) contain the aliases (lg n bits long) for the values stored in ONR and PNR, respectively. The AT enables the MMU to store a smaller tag with each address instead of a full object/process name.
The structure of a TLB entry is shown in Figure 10. Each entry consists of a tag of size \( \log a \) bits, a virtual page number (VPN) of size \( v \) bits, a physical page number (PPN) of size \( p \) bits, and eight 1-bit flags. When the global flag (G) is set, the VPN is global (kernel virtual address) and will match any valid TLB entry that has the same VPN, regardless of the tag value. When the dirty flag (D) is set, it indicates that the page has been written into. The MMU generates a fault when a write is attempted and D is not set. This fault is necessary to enable updating of the corresponding page table entry in the software data structures maintained by the kernel. When the valid flag (V) is set it indicates that this TLB entry is valid. The two protection bits (PR) encode one of four possible page protection modes, as shown in Table 3. The history bit (H) is used by the MMU to implement a pseudo least-recently used algorithm [Mot86] that decides which TLB entry to replace when WriteTLBEntryLRU command is issued. The two software flags (S) can be set by the software and have no meaning for the MMU.

Each address generated by the CPU is viewed by the MMU as shown in Figure 11. The high-order two bits are used to select one of the four spaces: 0, P, \( k_u \), and \( k_p \). Address translation proceeds as shown in Table 4: Addresses from 0 space (high order two bits=00) are tagged with the contents of OAR. P space addresses (high order two bits=01) are tagged with the contents of PAR. Addresses from \( k_u \) sub-space (high order bits=10) are tagged with 0. Addresses from \( k_p \) sub-space (high order bits=11) are ignored by the MMU. A tagged address matches a valid TLB entry if (a) the tag and VPN of the generated address matches the tag and VPN of the TLB entry, or (b) the VPN of the generated address matches the VPN of the TLB entry and the G bit is set.

An implementation that uses our MMU ties a portion of the \( k_p \) sub-space directly to physical memory. The \( k_p \) sub-space is intended for use by those portions of the kernel that need not be mapped dynamically. Specifically, most of the kernel consists of code, constants, and static data that need not be remapped once loaded into memory. By eliminating the need to translate addresses from those portions of the kernel, the poor locality properties of system kernels observed by Clark and Emer [CE85] do not affect the TLB. A similar approach is taken by the MIPS MMU [DMM86]. The disadvantage of this approach is a loss of some degree of protection,
because kernel software can inadvertently overwrite read-only portions of the kernel. However, kernel software is expected to be correct, and we do not expect this to be a major disadvantage. In §5.4 we show how to utilize this feature in the Ra kernel.

The kernel modifies the ONR to map a new O space on object call/return and process switches, and modifies the PNR to map a new P space on process switches. When the ONR (PNR) is changed by the software, the MMU searches the AT for an entry with the same value as the ONR (PNR), and OAR (PAR) is set to this value. If no matching entry is found in the AT, an entry is reused, and all TLB entries with the same tag as the index of the chosen entry are flushed (entries with the G bit set are not flushed). The MMU flushes a TLB entry by clearing the V bit.

The software and MMU communicate through the memory mapped registers shown in Figure 12, using the command set summarized in Table 5. The kernel writes the arguments of a command into the appropriate registers, writes the command into CommandReg, then reads back output arguments (if any) from the appropriate registers. The ReadTLBEntry and WriteTLBEntry commands are used by the kernel to read/write TLB entries. WriteTLBEntryLRU is used to overwrite the the first invalid entry or the least-recently used entry. ProbeTLB is used by the kernel to search the TLB for an entry, given a virtual address. Note that if the G bit is set in the TLBentryReg, ProbeTLB ignores the value of the tag, and returns the index of any valid entry that matches the given VPN. ProbeAT returns the tag associated with the given process/object name. FlushTLB flushes all TLB entries with the given tag (and G=0). SetONR and SetPNR write a new value into ONR and PNR, respectively.

The MMU interrupts the CPU (via an interrupt line) in one of three cases: When an address does not match any entry in the TLB, a miss-fault is generated; when a protection violation is detected, a protection-fault is generated; and when a write is attempted to a page with the D bit cleared, a dirty-fault is generated. It is up to the kernel to distinguish the type of the fault by examining TLBentryReg and VirtAddrReg. We chose this simple mechanism to allow our MMU to be easily interfaced to a variety of microprocessors. On a miss-fault, the virtual address causing the fault is written into the VirtAddrReg. On a protection-fault or a dirty-fault,
the TLB entry of the requested page is written into TLBentryReg, and a value of \(0xffffffff\) is written into the VirtAddrReg. Therefore, if a value other than \(0xffffffff\) is read from the VirtAddrReg, the kernel discerns the fault to be a miss-fault. Otherwise, the D bit in the TLBentryReg determines whether the fault is a dirty-fault or a protection-fault. Note that the address \(0xffffffff\) is in \(k_p\) and can never generate a miss-fault because the MMU ignores all \(k_p\) addresses.

An implementation of the MMU has to fix the various parameters of the chip, such as the page size and number of TLB entries. We expect the MMU parameters to range over the following values: From our experience in building the Clouds kernel on the Vax architecture (page size=512 bytes) and the Ra kernel on the Sun-3 architecture (page size=8K bytes), we believe that a page size in the range 2K–4K bytes is an appropriate size. From a simulation study of our MMU design (see §5.5), an AT size in the range 16–32 entries, and a TLB size in the range of 128–256 entries are appropriate. Ten bits should be an appropriate size for object/process names, given that the kernel can use the index of the object/process descriptor as its name (see §5.4). Table 6 lists these values.

5.4 Implementation of Ra using proposed MMU

The proposed MMU is well-suited to the Ra kernel. The Ra software machine-independent structures that describe each virtual space (VSD) and segment (stable) are sufficient for representing virtual space mappings, and no hardware-dependent structures are needed. The kernel is required to access the MMU in the following cases:

- **Object invocation and process switches.** On object call/return, the ONR is set to the index of the VSD of the object to be mapped. Each VSD in Ra has an index [BK88], and this index can be used for the value for ONR. Similarly, on process context switches, PNR is set to the index of the VSD describing the P space to be mapped.

- **MMU faults.** On a miss-fault, the kernel searches the VSD (as described in [BK88]) to locate the mapping information for the faulting address. As part of the normal page-fault
handling, the kernel checks the request for validity, and may have to access a partition to bring the required page into memory. When the mapping information is available, it is written into the TLB using `WriteTLBentryLRU` command. On a protection-fault, the kernel has the VPN available in `TLBEntryReg`, and can decide on the proper action. On a dirty-fault, the kernel should set the D bit using `WriteTLBentry`, after setting the corresponding bit in the software physical-page descriptor [BK88]. Note that copy-on-write semantics can be implemented upon detection of dirty-faults.

- **Reuse of a VSD.** When the kernel reuses a VSD, it must issue a `FlushTLB` command to make sure any entries from the destroyed VSD are purged.

- **Modifying the mapping of a page.** When the mapping of a page is modified (e.g. on a page-out operation), the corresponding entries in the TLB mapping this page has to be flushed. The commands `ProbeTLB`, `WriteTLB`, and `ReadTLB` are provided for this purpose.

To exploit the $k_v$ and $k_p$ portions of the K space no modifications to the architecture of Ra is required. Only the *system-object loader* [BHK*89] that is responsible for loading system objects in the K space needs to be modified. The loader should be aware of the differences between $k_v$ and $k_p$, and any portion of the system that may be remapped dynamically has to reside in $k_v$.

### 5.5 Evaluation

The approach to handle TLB reloading in software is not novel. Others have advocated a similar software approach to handle address translation misses [DMM86,MLM*86,CGBG88]. Among the advantages cited for this approach is the simplification of circuitry, and saving in chip area [DMM86]. The saving in chip area offers more space for a larger TLB.

This approach enables the MMU to match the OS requirements by using the best suited page table format as dictated by the software. By narrowing the semantic gap between the MMU and the operating system kernel, the programming model of the system is not hindered by the underlying hardware. In addition, the virtual memory system in the kernel is considerably simplified.
The cost of handling TLB misses in software is not a dominating cost as may first seem. The cost of TLB refill in software on the MIPS CPU is around 1.7 $\mu$s [DMM86] (using an 8 MHz CPU and a simple 2-level page table). In comparison, the cost of a TLB refill in hardware on the Vax-11/780 is around 4.4 $\mu$s. Cheriton et al. [CGBG88] estimated the cost of servicing an address translation miss on the VMP multiprocessor (using a software page table structure similar to ours) to be 10 $\mu$s on an average, versus about 4 $\mu$s with a conventional page table. By their estimates, this scheme adds about 4.8 nanoseconds on an average to each memory reference, which is an acceptable overhead given that each memory access costs 75 nanoseconds on VMP [CGBG88].

Note that when a TLB miss-fault is a result of a missing page (i.e. the required page is not in main memory), the cost of bringing the missing page into memory overshadows the extra overhead of the software-controlled TLB (bringing a page into memory costs on the order of tens of milliseconds). Appropriately tagging the TLB entries, a large page size, and the use of $k_v$ and $k_p$ sub-spaces, help keep the TLB miss rate low (similar arguments are made in [DMM86]).

We evaluate the proposed MMU with respect to the criteria of §3:

C1. The MMU caches address translation information across object invocation and process context switches. By separately tagging O and P space addresses, translation information is kept in the TLB when O and/or P spaces are changed. We evaluated our scheme using trace-driven simulation, and compared its performance to the usual case of flushing the TLB on each object call and return [RK88]. The simulation study used synthetic address traces and measured the effect of caching O space addresses only. The results of our performance study is summarized below, where $T_1$ refers to our scheme, while $T_0$ refers to the usual case of flushing the TLB on object invocations:

- For a given TLB size, the miss ratio of $T_1$ is consistently less than the miss ratio $T_0$. For a TLB size of 128, the miss ratio of $T_1$ is 100 times less than the miss ratio of $T_0$.

- The $T_1$ miss ratio decreases as the TLB size increases beyond the point when the
TO scheme miss ratio levels off. T1 uses the additional TLB entries to cache more address translations that may be used in the future, unlike T0 which flushes cached translations on each object call and return.

- Depending on the ratio of $T_h/T_m$ (where $T_h$ is the hit cycle time, and $T_m$ is the miss cycle time), the overall percentage improvement of T1 over T0 ranges from around 15% to 30%.

C2. One operation is required to map an O space, and at most two operations are needed for a process context switch.

C3. The MMU does not impose a page table structure, and the kernel is able to use an efficient representation of virtual spaces.

C4. The MMU allows aliasing and no restrictions are imposed on sharing memory among spaces.

C5. Allocating a range of addresses does not require MMU intervention (though the software may preload TLB entries). Deallocating a range of addresses requires the kernel to flush all entries in the TLB in the deallocated range. Because there is no practical restriction on the number of segments attached to a virtual space, we expect the number of attach/detach operations to be small. We expect that attaching small segments for passing parameters and temporary manipulation of segments to be more frequent than attaching/detaching large segments. Using the ProbeTLB and WriteTLBentry, the kernel can detach a segment in a small number of operations (proportional to the number of pages in the window mapping the segment).

6 Support for Atomic Transactions

The requirements discussed so far are basic to managing object space. Systems such as Clouds [BHK*88] and Argus [LS82], support the notions of atomicity of computation and recoverability of data, sometimes referred to as transactions. This notion requires that memory segments be
recoverable if a computation needs to be aborted. Software techniques such as shadowing, and logging are usually employed to implement transactions [BH87]. However, it is possible to reduce the burden on the software by providing some mechanisms in the MMU for supporting transactions.

Some researchers have proposed hardware support for atomic transactions [Sto84, CM88]. IBM RT/PC implements one such mechanism called transaction locking in its MMU [IBM86]. The mechanism consists of adding lock bits and a transaction id to each page table entry. The lock bits represent read or read-write locks on individual lines of the page held by the transaction whose id appears in the entry. The software uses this mechanism to keep track of modified lines and to implicitly grant locks to transactions. Chang and Mergen [CM88] report on implementing a transaction system using these mechanisms. Their system uses the transaction locking mechanism to implement a wide range of serializability and atomic update properties.

A transaction locking mechanism similar to IBM RT/PC can be easily added to our MMU design. The addition consists of appending each TLB entry with a process name field and several lock bits. The use of such a mechanism to implement atomic transactions is an interesting research issue to pursue.

7 Conclusions

Shared memory is a simple and powerful paradigm for structuring systems. Object-based operating systems embody this paradigm as the basic building block in providing a clean separation between policies and mechanisms. The discussion in this paper reveals some of the inadequacies of commercial MMUs for supporting the requirements of object-based operating systems. The design presented in this paper shows that it is possible to meet these requirements with a combination of the features available in commercial MMUs.
References


Figure 1: Model of Computation

<table>
<thead>
<tr>
<th>SR</th>
<th>Assign to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>K space</td>
</tr>
<tr>
<td>6</td>
<td>one code segment of O space</td>
</tr>
<tr>
<td>5</td>
<td>stack segment of P space</td>
</tr>
<tr>
<td>4</td>
<td>changed dynamically to access O space segments</td>
</tr>
<tr>
<td>3</td>
<td>changed dynamically to access P space segments</td>
</tr>
<tr>
<td>2</td>
<td>external-to-internal name mappings for O segments</td>
</tr>
<tr>
<td>1</td>
<td>external-to-internal name mappings for P segments</td>
</tr>
<tr>
<td>0</td>
<td>scratch register</td>
</tr>
</tbody>
</table>

Table 1: Assignment of space registers in HP architecture

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>2</td>
<td>$2^{19}$ bytes O + P total size</td>
</tr>
<tr>
<td>TIA</td>
<td>3</td>
<td>4 entries for O space, 4 entries for P space</td>
</tr>
<tr>
<td>TIB</td>
<td>6</td>
<td>64 entries per level 2 table</td>
</tr>
<tr>
<td>TIC</td>
<td>8</td>
<td>256 entries per level 3 table</td>
</tr>
<tr>
<td>TID</td>
<td>0</td>
<td>No level 4 tables</td>
</tr>
<tr>
<td>PS</td>
<td>11</td>
<td>2K bytes pages</td>
</tr>
</tbody>
</table>

Table 2: Possible assignment of MC68851 parameters

<table>
<thead>
<tr>
<th>Value of PR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Kernel read-only access; user no access</td>
</tr>
<tr>
<td>01</td>
<td>Kernel read-write access; user no access</td>
</tr>
<tr>
<td>10</td>
<td>Kernel read-write access; user read-only access</td>
</tr>
<tr>
<td>11</td>
<td>Kernel read-write access; user read-write access</td>
</tr>
</tbody>
</table>

Table 3: Protection Bits
<table>
<thead>
<tr>
<th>Value of bits 31 &amp; 30</th>
<th>MMU Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Tag VPN with OAR and compare</td>
</tr>
<tr>
<td>01</td>
<td>Tag VPN with PAR and compare</td>
</tr>
<tr>
<td>10</td>
<td>Tag VPN with 0 and compare</td>
</tr>
<tr>
<td>11</td>
<td>Ignore address</td>
</tr>
</tbody>
</table>

Table 4: Address translation process

<table>
<thead>
<tr>
<th>Command</th>
<th>Input Argument</th>
<th>Output Argument</th>
<th>Side Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadTLBentry</td>
<td>IndexReg</td>
<td>TLBentryReg</td>
<td></td>
</tr>
<tr>
<td>WriteTLBentry</td>
<td>IndexReg, TLBentryReg</td>
<td></td>
<td>TLB entry updated</td>
</tr>
<tr>
<td>WriteTLBentryLRU</td>
<td>TLBentryReg</td>
<td></td>
<td>LRU TLB entry updated</td>
</tr>
<tr>
<td>ProbeTLB</td>
<td>TLBentryReg</td>
<td></td>
<td>If found, output in index field of IndexReg Otherwise, P is set</td>
</tr>
<tr>
<td>ProbeAT</td>
<td>NameReg</td>
<td></td>
<td>If found, output in tag field of t_{hi}. Otherwise, P is set</td>
</tr>
<tr>
<td>FlushTLB</td>
<td>tag field in TLBentryReg</td>
<td></td>
<td>All entries with same tag and G=0 are invalidated</td>
</tr>
<tr>
<td>SetONR</td>
<td>NameReg</td>
<td></td>
<td>ONR updated</td>
</tr>
<tr>
<td>SetPNR</td>
<td>NameReg</td>
<td></td>
<td>PNR updated</td>
</tr>
</tbody>
</table>

Table 5: Proposed MMU commands

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>128 – 256 entries</td>
</tr>
<tr>
<td>n</td>
<td>16 – 32 entries</td>
</tr>
<tr>
<td>a</td>
<td>10 bits</td>
</tr>
<tr>
<td>Page size</td>
<td>2K – 4K bytes</td>
</tr>
<tr>
<td>v</td>
<td>19 – 18 bits</td>
</tr>
<tr>
<td>p</td>
<td>18 bits</td>
</tr>
<tr>
<td></td>
<td>(0.5G – 1G bytes of physical memory)</td>
</tr>
</tbody>
</table>

Table 6: Proposed MMU parameters

37
Virtual Space defined by descriptor

Figure 2: Ra Virtual Space

Figure 3: Structure of a window
Figure 4: Address translation in HP Precision Architecture
Figure 5: Address translation in IBM RT/PC MMU

Figure 6: Sun-3 MMU
Figure 7: Use of virtual address to index page table in MC68851

Figure 8: Formation of Global Virtual Address in SPUR

Figure 9: CPU virtual address space of proposed MMU
Figure 10: Proposed MMU

Figure 11: CPU Virtual Address

Figure 12: Proposed MMU communication registers
distributed Operating Systems and Machine Architecture: Reducing the Semantic Gap

The focus of our work has been in exploring the interface between distributed operating system and the machine architecture, with an eye toward suggesting ways and means of reducing the semantic gap between the two. Operating systems structures for a distributed environment follow one of two paradigms: message-passing or object-invocation. Distributed message-based operating systems consist of a small message-passing kernel supporting a collection of system server processes that provide such services as resource management, file service, and global communications. For such an architecture to be practical, it is essential that basic messages be fast since they often replace what would be a simple procedure call or "kernel call" in a more traditional system. Distributed object-based operating systems view the resources of the system as a collection of objects. Loads is an object-based distributed operating systems research project at Georgia Tech. System services are encoded in objects and access to system services are requested by invocations on the appropriate system object. The speed of object invocation is often used as a yardstick for measuring the performance of object-based systems.

The work completed under this grant set out to identify the software and hardware mechanisms for efficiently mapping the two paradigms of operating system structures in a distributed setting. The software mechanisms identified included coherence maintenance algorithms for distributed shared memory, and a software architecture for the implementation of distributed shared memory; the hardware mechanisms identified included queueing mechanisms in the memory bus for supporting message-passing, the design and analysis of memory management structures for object-based systems, and the design and analysis of synchronization primitives for multiprocessor caches.
PART IV - SUMMARY DATA ON PROJECT PERSONNEL

The data requested below will be used to develop a statistical profile on the personnel supported through NSF grants. The information on this part is solicited under the authority of the National Science Foundation Act of 1950, as amended. All information provided will be treated as confidential and will be safeguarded in accordance with the provisions of the Privacy Act of 1974. NSF requires that a single copy of this part be submitted with each Final Project Report (NSF Form 98A); however, submission of the requested information is not mandatory and is not a precondition of future awards. If you do not wish to submit this information, please check this box.

Please enter the numbers of individuals supported under this NSF grant. Do not enter information for individuals working less than 40 hours in any calendar year.

<table>
<thead>
<tr>
<th>*U.S. Citizens/Permanent Visa</th>
<th>Pl's/PD's</th>
<th>Post-doctorals</th>
<th>Graduate Students</th>
<th>Under-graduates</th>
<th>Precollege Teachers</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hawaiian or Pacific Islander</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black, Not of Hispanic Origin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hispanic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>White, Not of Hispanic Origin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U.S. Citizens</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U.S. Citizens &amp; Non-U.S.</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Member of individuals to have a handicap at limits a major activity.

*Use the category that best describes person's ethnic/racial status. (If more than one category applies, use the one category that most closely reflects the person's recognition in the community.)

- **AMERICAN INDIAN OR ALASKAN NATIVE**: A person having origins in any of the original peoples of North America, and who maintains cultural identification through tribal affiliation or community recognition.
- **ASIAN OR PACIFIC ISLANDER**: A person having origins in any of the original peoples of the Far East, Southeast Asia, the Indian subcontinent, or the Pacific Islands. This area includes, for example, China, India, Japan, Korea, the Philippine Islands and Samoa.
- **BLACK, NOT OF HISPANIC ORIGIN**: A person having origins in any of the black racial groups of Africa.
- **HISPANIC**: A person of Mexican, Puerto Rican, Cuban, Central or South American or other Spanish culture or origin, regardless of race.
- **WHITE, NOT OF HISPANIC ORIGIN**: A person having origins in any of the original peoples of Europe, North Africa or the Middle East.

THIS PART WILL BE PHYSICALLY SEPARATED FROM THE FINAL PROJECT REPORT AND USED AS A COMPUTER SOURCE DOCUMENT. DO NOT DUPLICATE IT ON THE REVERSE OF ANY OTHER PART OF THE FINAL REPORT.
Distributed Operating Systems and Machine Architecture: Reducing the Semantic Gap
Final Report for NSF Project MIP-8809268

Umakishore Ramachandran
College of Computing
Georgia Institute of Technology
Atlanta, Ga 30332-0280
Phone: (404) 894-5136
E-mail: rama@cc.gatech.edu
The final report summarizes the results of the research carried out under the NSF grant MIP-8809268.

The work proposed under the grant was the investigation of operating systems structures for distributed environments, with an eye toward reducing the semantic gap between the machine architecture and the operating systems. The specific work carried out under the grant may be categorized in three broad categories:

1. The design and implementation of a software architecture for supporting the operating system abstractions in a distributed object-based system.

2. The specification and analysis of hardware assists for enhancing the performance of both message-based and object-based operating system structures.

3. The specification, design, and analysis of synchronization mechanisms in hardware for assisting the efficient design of software systems in shared memory multiprocessors.

The NSF grant had provision for and was used for supporting the salaries of the PI, and one Ph. D. student: Joonwon Lee. A second Ph. D. student, Yousef Khalidi, (not directly paid from this grant, but paid from an institutional infrastructure grant from NSF) also worked with the PI in many of the issues addressed by this grant. The work carried out under this grant produced two Ph. D. theses, and the abstracts from these theses have been enclosed. Yousef Khalidi graduated with a Ph. D. in computer science in April 1989, and is currently working with Sun Microsystems. Joonwon Lee was supported for the entire period of the grant, and is graduating with a Ph. D. in computer science at the end of May. Upon graduation, he will be joining IBM Research Labs, Endicott, New York. It should be pointed out that Joonwon Lee has been supported by an NSF PYI award (September 1990 to present) since the expiration of this grant that was awarded to the PI in 1990.

We have presented the results from this research at several prestigious conferences (including International Symposium on Computer Architecture, and International Conference on Parallel Processing), research labs (including IBM Almaden Research Center, IBM Research in Yorktown Heights, Digital Equipment Corporation in Boston, and the Center for Development of Advanced Computing in India), and Universities (including University of Wisconsin - Madison, and University of Southern California, Los Angeles).

We briefly highlight the significant results from the work carried out under the grant. Following this a list of publication resulting from the grant is given. A set of selected publications and cover pages from all the publication resulting from this grant is also enclosed.
Distributed Shared Memory

Recently, the notion of "Distributed Shared Memory" has received considerable attention. It provides the abstraction of shared memory in a non-shared memory (distributed) architecture. Our contribution to this area has been in unifying the management of distributed shared memory with process synchronization. We proposed new algorithms for coherence maintenance of distributed shared memory [RAK89], an implementation of these algorithms in the Clouds distributed operating system [RK89c] to serve as the basic mechanism for page transport in the system, techniques for programming with distributed shared memory [RK89d], and low-level mechanisms for addressing fault-tolerance in distributed shared memory systems [MR89]. The key idea in our algorithms are the exploitation of application semantics in terms of synchronization information to reduce the overhead of coherence maintenance.

Hardware Support for Object-Based Systems

Object-based systems such as Clouds are very heavily reliant on efficient virtual memory management techniques. This in turn depends on the support that hardware memory management units provide as well as how efficiently remote page faults are serviced through the distributed shared memory facility that we mentioned earlier. Through an exhaustive measurement of the implementation we identified the impediments to the performance of object-based systems [RK89a], and suggested architectural solutions to these impediments. The main problem limiting the efficient implementation of object invocation is the insufficient support provided by hardware memory management units for modern operating systems abstractions. We proposed a set of criteria that memory management units ought to have [RK88], evaluated several commercial memory management units with respect to these criteria, and proposed a design of a memory management unit that satisfies these criteria [RK89b].

Shared Memory Multiprocessors

A key to designing software structures for parallel processors, requires an understanding of the capabilities of the underlying architecture. By the same token, the design of hardware mechanisms should exploit the semantic information forthcoming from the system software such as the operating system and the language runtime system. In this vein, we identified the features of large-scale shared memory multiprocessors that are important from the point of view of realizing parallel applications [LR91]. In particular, we argue for relaxing the memory model seen
by the programmer, and providing efficient synchronization mechanisms in hardware. Further, we identified the performance advantage of providing such synchronization support in hardware [LR90, RL88].

**Hardware Support for Interprocess Communication**

Lastly, the paper reference [RSV90] identifies hardware mechanisms for efficiently supporting message passing. While most of the work reported in [RSV90] was done prior to this grant period, it is appropriate to include this work in this final report since the PI had to do a sizable amount of work in extending his Ph. D. dissertation work to this final form.

The following is a list of the publications that resulted from this grant, and referred to in the final report.

**References**


Thesis Abstracts
Hardware Support for Distributed Object-based Systems

A Thesis
Presented to
The Academic Faculty

By

M. Yousef Amin Khalidi

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy
in the
School of Information and Computer Science

Georgia Institute of Technology
June 1989
Summary

The object-based approach is an attractive model for structuring distributed systems. Distributed object-based operating systems view the resources of the system as a collection of objects. The primary mechanism in object-based systems is the mapping of the invoked object into the invoking thread's address space. Hence the performance of object-based systems depends crucially on the efficiency of this memory mapping. The problem gets exacerbated with distribution since the invoked object may now be located on a remote node. This research concentrates on identifying local and network-wide memory management requirements and on providing software and hardware mechanisms to handle these requirements.

The contributions of this research are twofold. In the area of local memory management, this research reveals some of the inadequacies of commercial memory management units (MMUs) for supporting the requirements of object-based operating systems, and presents the design of an MMU that meets these requirements. The design presented in this thesis shows that it is possible to meet these requirements with a combination of the features available in commercial MMUs.

In the area of network-wide memory management, this work suggests structuring the distributed operating system around the concept of distributed shared memory. Since a thread can potentially invoke any object, the virtual address spaces of all objects can be viewed as constituting a global distributed shared memory. Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-mapping mechanism. An organization and mechanisms for
supporting this abstraction of a distributed shared memory are presented, and a software implementation is described. These mechanisms serve as the backbone for implementing object invocation, synchronization mechanisms, paging of memory segments, and process and object migration in object-based kernels.
Architectural Features for Scalable Shared Memory Multiprocessors

A Thesis
Presented to
The Faculty of the Division of Graduate Studies

By

Joonwon Lee

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy
in Information and Computer Science

Georgia Institute of Technology
May 1991
Cache memories are important in multiprocessors because of memory latencies caused by bus saturation and/or multi-hop interconnections. Traditional multiprocessor cache protocols treat synchronization accesses the same way as normal read/write memory accesses. This approach leads to inefficiencies in performing synchronization operations which ultimately limits the scalability of such systems. We propose synchronization schemes in multiprocessors using cache memories: one for shared bus, and another for general interconnection networks. Each cache line maintains states for synchronization as well as for cache coherence, and the cache protocol ensures correct synchronization operations.

Providing the ability to distinguish between synchronization requests and read/write memory accesses allows the software to defer global coherence for shared data until it is absolutely necessary. In this weak consistency model, strong memory coherence may be enforced only at synchronization point. We propose new cache primitives for the weak consistency model and show how they can be implemented with private caches.

To assess the performance gain due to our cache protocols, we have performed simulation studies. The workload model represents a dynamic scheduling paradigm which is the kernel of several parallel programs. There is a global work queue of tasks that represents the parallel computation. Each processor takes a task from the work queue and executes it. Queue accesses from the processors are mutually exclusive and
barrier synchronizations at the end of a phase of computation are used when needed. Results from simulation studies show that our protocol performs significantly better than the traditional approaches.
Cover Pages of Publications
Synchronization with Multiprocessor Caches *

Joonwon Lee    Umakishore Ramachandran
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332 USA
e-mail: joon@sybil.gatech.edu, rama@gatech.edu

Abstract

Introducing private caches in bus-based shared memory multiprocessors leads to the cache consistency problem since there may be multiple copies of shared data. However, the ability to snoop on the bus coupled with the fast broadcast capability allows the design of special hardware support for synchronization. We present a new lock-based cache scheme which incorporates synchronization into the cache coherency mechanism. With this scheme high-level synchronization primitives as well as low-level ones can be implemented without excessive overhead. We derive cost functions for well-known synchronization methods, and our lock-based scheme. To accurately predict the performance implications of the new scheme, a new simulation model is developed embodying a widely accepted paradigm of parallel programming. It is shown that our lock-based protocol outperforms existing cache protocols.

1 Introduction

Cache memories have been used to reduce memory access latency in uniprocessors. In bus-based shared memory multiprocessors they may additionally reduce bus contention. However, private caches introduce the cache coherence problem [8]. The shared bus enables each cache controller to monitor the bus traffic and initiate appropriate actions to keep the shared data coherent. A group of cache coherence schemes called snooping cache protocols use this feature [13, 16, 18, 19, 24]. They are implemented in hardware and may not be visible to the programmer.

Most snooping cache protocols (Section 2) do not take into account synchronization requests that usually precede shared data accesses. Therefore, strong coherence among multiple copies is blindly enforced resulting in possibly unnecessary bus traffic for invalidation and data transfer. Usually, access to shared data is acquired via synchronization methods such as locks, semaphores, and barriers. Thus there is additional delay in accessing the synchronization variables and then acquiring the actual data.

In this paper, we present a cache coherence protocol supporting lock primitives (Section 3). Our scheme utilizes locking information provided by the software (e.g. compiler) to distinguish between shared-locks and exclusive-locks. We construct a distributed hardware-assisted FIFO queue of processors waiting for a given lock. With this scheme, the cache mechanism emerges as a visible part of the architecture since programmers should understand it to develop efficient parallel programs.

Efficient interprocessor synchronization and mutual exclusion are imperative to assure good performance for parallel programs. Therefore, we must evaluate the synchronization efficiency of cache protocols (Section 4).

Evaluating a multiprocessor system is a challenging task. Trace driven simulation has been used for multiprocessor evaluation [1, 11, 12, 25], but tracing parallel programs has so far been restricted to a small number of processors. Furthermore, the trace is affected by the host multiprocessor's architectural characteristics such as cache protocol, synchronization primitives, and interconnection network. As Bitar [4] points out using traces generated by software makes it difficult to verify the validity of the predicted results. Therefore, we have developed a new method for the evaluation of cache protocols. Our simulation model and some results are presented in Section 5.

2 Snooping Caches

Hardware cache coherence schemes for shared-bus multiprocessors have evolved into two categories, namely, invalidation schemes and write update schemes.

In invalidation schemes [13, 16], a write to a cached line results in invalidating copies of this line present in other caches. If writes to cached data always trigger invalidations, the bus is easily saturated with even a few participating processors [27]. To reduce the invalidation rate, Goodman developed the write once protocol [13] in which only the first write to a cached data updates main memory, and is used as a cue by other caches for invalidating their own copies. The Berkeley protocol [16] assumes an invalidation line on the bus to explicitly invalidate peer caches. Since an invalidation is induced even with a first
Locks, Directories, and Weak Coherence - A Recipe for Scalable Shared Memory Multiprocessors*

(FULL PAPER ENCLOSED UNDER "SELECTED PUBLICATIONS IN FULL")

Joonwon Lee Umakishore Ramachandran

Technical Report GIT-CC-90/68
December 1990

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

joon@cc.gatech.edu, rama@cc.gatech.edu

Abstract

Bus based multiprocessors have the limitation that they do not scale well to large numbers of processors due to the bus becoming a bottleneck with the current bus technology. Lock-based protocols have been suggested as a possible way of mitigating this bottleneck for single bus systems with snooping ability. In this research, we are interested in extending lock-based protocols to general interconnection networks. Directory based cache coherence schemes have been proposed for such networks. We are investigating a combination of locking with directory based schemes. Further, most protocols in the literature until now, assume a strong coherence requirement. However, recent research has shown that it is possible to weaken this coherence requirement. Such an approach is expected to reduce the coherence overhead even further, making it an appealing one for building scalable systems.

* This work is supported in part by NSF grant MIP-8809268.
Implementing Fault-tolerant Atomic Transactions Using Distributed Shared Memory*

Ajay Mohindra    Umakishore Ramachandran

School of Information and Computer Science
Georgia Institute of Technology

Abstract

The abstraction for supporting the notion of shared memory in a non-shared memory (distributed) architecture is referred to as distributed shared memory. We have implemented a set of mechanisms for maintaining the coherence of distributed shared memory. In this paper we show how distributed fault-tolerant atomic transactions can be implemented elegantly using these mechanisms. Since our mechanisms combine the tasks of synchronization and data transfer into one, transaction-level locking of segments comes for free. Fault-tolerance is achieved by using replication on top of distributed shared memory. Atomicity of transactions is guaranteed through use of a two-phase commit protocol. We show that the resulting implementation incurs much less overhead compared to message-passing schemes.

*This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.
Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer

Umakishore Ramachandran
Mustaque Ahamad
M. Yousef A. Khalidi

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Ga 30332-0280 USA

Abstract,

Clouds is a distributed operating systems research project at Georgia Tech. With threads and passive objects as the primary building blocks, Clouds provides a location-transparent protected procedure-call interface to system services. Mechanisms for synchronization within objects, and atomicity of computation are supported in Clouds. The primary kernel mechanism in object-based systems such as Clouds is the mapping of the object into the address space of the invoking thread. Hence the performance of such systems depends crucially on the efficiency of memory mapping. The problem gets exacerbated with distribution since now the object invoked by a thread may be located on a remote node. Since a thread can potentially invoke any object, the virtual address spaces of all objects can be viewed as constituting a "global distributed shared memory". Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-mapping mechanism. We present an organization and mechanisms for supporting this abstraction of a distributed shared memory. We propose a distributed shared memory controller that provides mechanisms for efficient access and consistency maintenance of the distributed shared memory. The novel feature of our approach is the exploitation of process synchronization to simplify consistency maintenance. The distributed shared memory mechanisms serve as the backbone for implementing object invocation, synchronization mechanisms, and network-wide memory management in the Clouds system.

1 Introduction

We are exploring hardware support to improve the performance of object-based distributed operating systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network. There are one or more processors and a certain amount of memory in each node. Nodes do not share memory; message exchange across the network is the only mechanism for communication between them.

Many operating systems designs for such an environment [4,14,20,32] place a message-passing kernel on each node, supporting processes and communication between them via explicit messages. This kernel supports both local communication—communication between processes on the same node—and non-local or remote communication (sometimes implemented via a distinguished network manager process). Access to system services are requested via protected procedure calls in a traditional system, whereas in a message-based operating system they are requested via message passing. While a simple procedure call costs just a few instructions, and a protected procedure call (kernel call) costs a few hundred instructions, IPC costs a few thousand instructions in several systems that we studied [28]. Message-based operating systems are attractive for structuring distributed systems due to the separation of policy (encoded in server processes) from mechanism (in the kernel).

Object-based distributed operating systems [7,2,39,34,26] view the resources of the system as a collection of objects. Clouds [7] is an object-based distributed operating system being developed at Georgia Tech. In Clouds, system services are encoded in passive objects (syntactic units that are similar in flavor to the server processes of message-based systems) that occupy distinct virtual address spaces in the system. Access to system services are requested by invocations (similar to the protected procedure calls of traditional systems) into the appropriate system object. The speed of object invocation is often used as a yardstick for measuring the performance of object-based systems. In passive object-based systems, invocation performance depends on the efficiency of object memory management (see §2). The problem gets exacerbated with distribution since now the invoked object may be located on a remote node.

In this paper we suggest mechanisms (that can be implemented in hardware/firmware) for supporting the abstraction of "globally distributed shared memory". In §2, we give the relevance of our work and motivate the need for supporting this abstraction. Related work in the area of memory coherence is presented in §3. With the Clouds operating system as our target application, we present our ideas on customizing the memory coherence requirements in §4. Our proposed hardware organization, the primitives provided by the distributed shared memory controller, and the algorithms for maintaining the consistency of the distributed shared memory are discussed §5. In §6 we describe a software implementation of the proposed primitives. A performance evaluation of our scheme is presented in §7. Finally, our conclusions are presented in §8.

2 Relevance

Clouds [7] is a distributed operating system that is intended to provide a unified environment over distributed hardware. Location independence for data as well as processing, atomicity of distributed computation, and fault-tolerance are some of the research goals of Clouds.

Objects and threads are the basic building blocks of Clouds. Objects are passive entities and specify a distinct and disjoint piece of the global virtual address space that spans the entire network. An object is the encapsulation of the code and data needed to implement the entry points in the object. Thus a Clouds object can be considered syntactically equivalent to an abstract data type in the programming language parlance. Access to entry points in the object are accomplished through a capability mechanism in the kernel.

Threads are the only active entities in the system. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object. A thread enters an ob-
An Evaluation of Memory Management Structures for Object-based Systems*

M. Yousef Amin Khalidi  
Sun Microsystems, Inc.  
2550 Garcia Ave., MS 5-44  
Mt. View, CA 94043 U.S.A.  
Tel. (415) 336-5392  
e-mail: yak@eng.sun.com

Umakishore Ramachandran  
College of Computing  
Georgia Institute of Technology  
Atlanta, Ga 30332-0280 U.S.A.  
Tel. (404) 894-5136  
e-mail: rama@cc.gatech.edu

Abstract

Object-based operating systems have the desirable property of separating policies from mechanisms, while providing a protected procedure call interface for accessing system services. However, the kernel mechanisms in such systems rely very heavily on efficient memory management. A set of criteria for supporting the kernel mechanisms in object-based systems is presented. Five commercial memory management units (MMUs) are surveyed with respect to these criteria. We then present the design of a simple MMU tailored for object-based systems. The proposed design is an engineering solution combining the features available in commercial MMUs.

Keywords: Operating systems, object-based systems, memory management, object invocation.

*This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.
A Measurement-based Study of Hardware Support for Object Invocation

UMAKISHORE RAMACHANDRAN AND M. YOUSEF AMIN KHALIDI
School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332-0280, U.S.A.

SUMMARY

The object invocation paradigm is attractive for structuring distributed systems. Distributed object-based operating systems view the resources of the system as a collection of objects. Object invocation is the primary mechanism in such systems, and is often used as a yardstick for measuring the system performance. However, existing systems of this flavour exhibit poor performance due to the mismatch between the requirements of the object invocation mechanism and the machine architecture. Through measurements of an existing object-based kernel, we present a breakdown of the costs involved in implementing the object invocation mechanism. The measurements suggest architectural solutions to improve the performance of such systems. We present our preliminary studies towards providing hardware support for the object invocation mechanism.

KEY WORDS Object invocation Measurements Hardware support Distributed operating systems

INTRODUCTION

This paper explores architectural support to improve the performance of object-based distributed operating systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network. There are one or more processors and a certain amount of memory in each node. Nodes do not share memory; message exchange across the network is the only mechanism for communication between them.

Operating system structures for a distributed environment follow one of two paradigms: message passing or object invocation. Message-based operating systems place a message-passing kernel on each node, supporting processes and communication between them via explicit messages. This kernel supports both local communication—communication between processes on the same node—and non-local or remote communication, sometimes implemented via a distinguished network manager process. In a traditional system such as Unix, access to system services is requested via protected procedure calls, whereas in a message-based operating system it is requested via message passing. Message-based operating systems are attractive for structuring distributed systems due to the separation of policy, encoded in server processes, from mechanism in the kernel.
CONCLUSIONS AND FUTURE WORK

The Clouds kernel served well as a prototype, and enabled us to gain important insight into the requirements of the object-based model of computation. However, it has several drawbacks. It is a monolithic kernel that is hard to modify and maintain, and the implementation is very machine-dependent. In addition, message transmission in Clouds is slow. Sending messages from one process running in a local kernel to a process running in a remote kernel takes roughly 10 ms. Most of this cost can be attributed to a poor network interface.\textsuperscript{38} Other systems, such as QuickSilver\textsuperscript{1} and V,\textsuperscript{1} report numbers in the range of 6 to 1.5 ms on faster hardware.

Our notions on structuring object-based operating systems has matured since the prototype design was begun. A new kernel for Clouds, called the Ra kernel, has been designed and implemented.\textsuperscript{5,9} Ra runs on the Sun-3 architecture, and incorporates support for distributed shared memory. The DSMC has been implemented in software and we are currently in the process of evaluating the implementation.\textsuperscript{36,37} Further refinement and implementation of an MMU tailored to object-based systems that incorporates our TLB scheme, and a hardware module that implements the DSMC protocol are some of the work we have identified for future research.

ACKNOWLEDGEMENT

This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.

REFERENCES

A Design of a Memory Management Unit for Object-based Systems

Umakishore Ramachandran M. Yousef Amin Khalidi

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Ga 30332–0280 USA

Abstract

Object-based operating systems have the desirable property of separating policies from mechanisms, while providing a protected procedure call interface for accessing system services. However, the kernel mechanisms in such systems rely very heavily on efficient memory management. A set of criteria for supporting the kernel mechanisms in object-based systems is presented. We then present the design of a simple MMU tailored for object-based systems. The proposed design is an engineering solution combining the features available in commercial MMUs.

1 Introduction

Traditional systems such as Unix, provide a protected procedure call interface for system services. However, in such systems the kernel tends to be monolithic and bulky since the policies and the mechanisms are all embedded in the kernel. In recent times, especially with the advent of local area networks, message-based operating systems have become popular [8,1,21,11]. Policies are encoded in server processes and the kernel is lean and supports simply a message passing mechanism between processes. System services are requested by sending and receiving messages. However, the major drawback in such systems is the fact that a round-trip message is involved for requesting system services which can be an order of magnitude more expensive than a protected procedure call [18].

Object-based systems [5,2,14,17] view the resources of the system as a collection of objects. An object is the encapsulation of the code and data needed to implement the entry points in the object. These entry points provide the procedural interface for an activity to execute the code in the object. System services are requested via invocations (similar to a protected procedure call) on these entry points. At the same time the kernel is lean and supports simply a message passing mechanism between processes. System services are requested by sending and receiving messages. However, the major drawback in such systems is the fact that a round-trip message is involved for requesting system services which can be an order of magnitude more expensive than a protected procedure call [18].

Object-based systems [5,2,14,17] view the resources of the system as a collection of objects. An object is the encapsulation of the code and data needed to implement the entry points in the object. These entry points provide the procedural interface for an activity to execute the code in the object. System services are requested via invocations (similar to a protected procedure call) on these entry points. At the same time the kernel is lean and supports simply a message passing mechanism between processes. System services are requested by sending and receiving messages. However, the major drawback in such systems is the fact that a round-trip message is involved for requesting system services which can be an order of magnitude more expensive than a protected procedure call [18].

Object-based systems [5,2,14,17] view the resources of the system as a collection of objects. An object is the encapsulation of the code and data needed to implement the entry points in the object. These entry points provide the procedural interface for an activity to execute the code in the object. System services are requested via invocations (similar to a protected procedure call) on these entry points. At the same time the kernel is lean and supports simply a message passing mechanism between processes. System services are requested by sending and receiving messages. However, the major drawback in such systems is the fact that a round-trip message is involved for requesting system services which can be an order of magnitude more expensive than a protected procedure call [18].

Clouds [5] is an example of such an object-based system. Ra [4] is an operating system kernel designed to be the nucleus of Clouds. The abstractions that Ra uses for supporting memory management are segments and virtual spaces. A Ra virtual space is described by a segment called the virtual space descriptor (VSD) that contains a collection of windows. Each window is a data structure that identifies the segment that backs a range of addresses in the virtual space (Figures 1 and 2).

An object virtual space is called an O space while a process virtual space is called a P space. Ra views the machine address space as consisting of three three distinct regions that are called K, O, and P hardware spaces for kernel, object, and process, respectively. The kernel is always mapped in the K hardware space. A process P space is mapped (or installed) into the P hardware space and unmapped on context switch. On object call, the O space of the invoked object is installed, and on object return, the O space of the caller object is installed.

Ra advocates a model of programming with a large number of possibly small-sized segments. Current implementation of Ra on the Sun-3 allows around 100 windows.

2 Comparison Criteria

As can be seen from the description of Clouds and Ra, an efficient implementation relies very heavily on effective management of virtual memory. In particular, the following five MMU features are required to support an efficient implementation of Clouds/Ra:

C1. The ability to cache TLB information across object invocation. The cost of flushing TLB entries at object call/return is an implicit cost that affects mostly user mode time. Systems with one large virtual address space do not require flushing the TLB across object invocations. Examples of such systems include the IBM RT/PC, HP Precision Architecture, and SPUR machine [22].

C2. The ability to implement an object call/return by performing a small number of operations on the MMU. The cost of required MMU operations is an explicit part of object invocation. This cost is quantified by the number of MMU registers and page table entries that need to be modified on each object call/return. In our proposed MMU (see §3), changing one register is the only operation that is required to effect an object call/return. A paper design of object invocation implementation on segmented single virtual address space systems, such as IBM RT/PC and HP Precision Architecture, reveals that only a small number of MMU operations are required [20]. However, our experience in implementing Ra on the Sun-3 MMU [4] reveals that many MMU operations are needed on this machine.

C3. The ability to represent sparse address spaces efficiently. Inefficient representation of sparse address spaces results in page tables that have large memory requirements and are sparse.
Abstract

Shared memory is a simple yet powerful paradigm for structuring systems. Recently, there has been an interest in extending this paradigm to non-shared memory architectures as well. For example, the virtual address spaces for all objects in a distributed object-based system could be viewed as constituting a global distributed shared memory. We propose a set of primitives for managing distributed shared memory. We present an implementation of these primitives in the context of an object-based operating system as well as on top of Unix.

1 Introduction

Programming with shared memory is well-understood and despite the interest in distributed and parallel systems for reasons of availability, fault-tolerance, and increased computational power, the style of programming these systems has not changed drastically. Even in non-shared memory architectures researchers have proposed a style that presents to the programmers an abstraction of a logical shared memory [19, 14, 8, 23]. Other researchers have proposed algorithms for maintaining the consistency of such a logically shared memory in non-shared memory architectures [17, 18, 21]. The abstraction for supporting the notion of shared memory on a non-shared memory (distributed) architecture is referred to as distributed shared memory (DSM) in this paper.

A second motivation for DSM is the current trend in structuring distributed systems using a collection of diskless computational servers, namely workstations, and a few data servers or file servers. In such an environment the code and data for program execution has to be paged-in from the data server. There are two issues here: The first one is a scheduling decision of 'where' to execute the program, one that is best left to a higher level policy making entity. The second one is the chore of bringing in the required data and code, i.e., remote paging. If sharing is coupled with this second issue, then we see that DSM presents itself as a natural facility for combining the two.

Several other researchers have proposed software architectures based on the shared memory paradigm, in different settings:

*This work has been funded in part by NSF grants CCR-8619886 and MIPS-8809268.
An Implementation of
Distributed Shared Memory*

Umakishore Ramachandran
School of ICS
Georgia Institute of Technology
Atlanta, Ga 30332
Phone: (404) 894-5136
E-mail: rama@gatech.edu

M. Yousef A. Khalidi
Sun Microsystems
2550 Garcia Avenue
Mountain View, Ca 94043
Phone: (415) 336-5392
E-mail: yak@sun.com

Abstract

Shared memory is a simple yet powerful paradigm for structuring systems. Recently, there has been an interest in extending this paradigm to non-shared memory architectures as well. For example, the virtual address spaces for all objects in a distributed object-based system could be viewed as constituting a global distributed shared memory. We propose a set of primitives for managing distributed shared memory. We present an implementation of these primitives in the context of an object-based operating system as well as on top of Unix.

Keywords: Distributed shared memory, distributed operating systems, object-based systems.

*This work has been funded in part by NSF grants CCR-8619886 and MIPS-8809268. An earlier version of this paper was presented at the USENIX workshop on experiences in building distributed and multiprocessor systems, Ft. Lauderdale, Florida, October 1989.
Programming with
Distributed Shared Memory

Umakishore Ramachandran M. Yousef Amin Khalidi

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Ga 30332-0280 USA

Abstract
In a distributed system, remote services may be provided by either remote procedure call (RPC) mechanism, or by paging in the required memory segments and performing the services locally. The latter approach, termed Distributed Shared Memory (DSM) has several benefits given the current trend of structuring computing systems using diskless computational servers (workstations) and data servers (file servers). We propose a set of distributed shared memory mechanisms that handle network-wide memory management for an object-based system. We discuss the implementation of these mechanisms and provide examples of their use in implementing the programming language Linda, process migration, two-phase commit, and a distributed game.

1 Introduction
The resources in a loosely-coupled distributed system can be viewed as a collection of objects. The name space of all objects constitute a "global distributed shared memory". In this paper, we suggest mechanisms for supporting the abstraction of a globally distributed shared memory, and applications using these mechanisms. These mechanisms were designed in the context of the Clouds [7,13] object-based distributed operating system.

The paper is organized as follows: We start with an overview of Ra—a kernel for Clouds and its relation to the shared memory module which we call distributed shared memory controller (DSMC). In §3 we discuss related work, and in §4 we discuss the status of our implementation. The three sections that follow present applications that use the DSMC mechanisms in implementing the Linda parallel programming language [15], process migration, and 2-phase commit. Section 8 investigates the use of weak memory coherence semantics in the implementation of a distributed game. Section 9 evaluates the use of distributed shared memory vs. remote procedure call for invoking remote objects. Finally, conclusions and future work follow in §10.

2 Ra and DSMC
Ra [5,7,6] is an operating system kernel designed to be the nucleus of Clouds operating system. It currently runs on the Sun-3 architecture. Ra defines and manages three primitive abstractions: segment, virtual space, and isiba. Segments serve as containers of data and may be viewed as uninterpreted sequences of bytes. The contents of a segment may only be accessed when the segment is attached to a range of virtual addresses. Segments persist until explicitly destroyed. Each segment resides in a partition that is responsible for providing backing store for the segment. A partition is an entity that realizes, maintains, and manipulates segments (see §4).

Virtual spaces abstract the notion of an addressing domain. A Ra virtual space is described by a segment called the virtual space descriptor that contains a collection of windows. Each window is a data structure that identifies the segment that backs a range of addresses in the virtual space. A virtual space is composed (or built) by a sequence of attach operations, each of which defines one of the virtual space's windows. An object is a virtual space (called O space) that consists of code and data segments. The code segment of an O space has entry points that can be invoked by user processes.

Ra isibas are an abstraction of the fundamental notion of computation or activity and can be thought of as lightweight processes. Isibas may be used as daemons within the kernel or they may be associated with a virtual space (a P space) to implement a user process.

The machine virtual address space consists of three distinct regions that are called K, O, and P hardware spaces for kernel, object, and process, respectively. The kernel is always mapped in the K hardware space. (System objects, which we discuss in §6, are also mapped into the K space, but may be installed and removed dynamically.) A Process is mapped into the P hardware space and unmapped on context switch. A process consists of an isiba and a virtual space (P space). The object in which a process is currently executing is mapped into the hardware O space.

Local object invocation involves mapping the required memory segments of the object into the address space of the invoking process by installing the object as the current O space with the process's P space. The current trend in structuring distributed systems is to use a collection of diskless computational servers (workstations) and a few data servers (file servers). In such an environment, the code and data for the (local) invocation has to be paged-in from the data server. Further, for remote object invocation we have one of two choices: The first choice is to perform the computation at the node where the object resides (remote procedure call). The second choice is to make the invocation appear local by bringing in the segments required for the invocation. While we have to support the former for immovable objects (such as an object that reads disk blocks), we believe that the latter may be a better choice for movable objects. There are two reasons to support this belief:

- the principle of locality [14] that suggests an invocation (or other invocations in the same object) may be repeated
- the reduction in computational overhead due to the elimination of slave process management to support remote invocation at the node where the object resides [19].

Each host has a distributed shared memory controller (DSMC) that together with the network interface assists the host in mapping remote memory segments into virtual address space on the local host. The system memory is (logically) partitioned...
Processor Initiated Sharing in Multiprocessor Caches *

Umakishore Ramachandran
Joonwon Lee

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332 USA
Ph: (404) 894-5136

rama@gatech.edu, joon@pyr.gatech.edu

Abstract

We present a lock-based multiprocessor cache consistency protocol. This protocol requires the processes to provide information about the data they access: read or write, shared or non-shared. This information enables caches to avoid frequent invalidations and data transfers. Shared data can be accessed only through lock operations. These lock operations entail queues for the processes awaiting a lock. A distributed hardware-assisted queue which ensures fairness and efficient waiting scheme is proposed. Implementation of the protocol with a current computer bus is discussed.

Keywords: Shared Memory Multiprocessor, Cache Coherency, Process Synchronization

*This work is supported in part by NSF grant MIP-8809268
Hardware Support for Interprocess Communication

UMAKISHORE RAMACHANDRAN, MARVIN SOLOMON, AND MARY K. VERNON, MEMBER, IEEE

Abstract—In recent years there has been increasing interest in message-based operating systems, particularly in distributed environments. Such systems consist of a small message-passing kernel supporting a collection of system server processes that provide such services as resource management, file service, and global communications. For such an architecture to be practical, it is essential that basic message exchanges be fast, since they often replace what would be a simple procedure call or "kernel call" in a more traditional system. Careful study of several operating systems shows that the limiting factor, especially for small messages, is typically not network bandwidth but processing overhead. Therefore, we propose using a special-purpose coprocessor to support message passing. Our research has two parts. First, we partitioned an actual message-based operating system into computation and communication parts, executing, respectively, on a host and a message coprocessor interacting through shared queues, and measured its performance on a multiprocessor. Second, we designed hardware support in the form of a special-purpose smart bus and smart shared memory and demonstrated the benefits of these components through analytical modeling using Generalized Timed Petri Nets. Our analysis shows good agreement with the experimental results and indicates that substantial benefits may be obtained from both the partitioning of the software between the host and the message coprocessor and the addition of a small amount of special-purpose hardware.

Index Terms—Architecture support, bus protocol, distributed systems, measurements, message-based operating systems, performance Petri nets, system architecture.

I. INTRODUCTION

We are interested in providing hardware support to improve the performance of distributed systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network (LAN). There are one or more processors and a certain amount of memory in each node. The nodes do not share memory; message exchange across the network is the only mechanism for communication between them. Many recent operating systems designs for such an environment [6], [25]-[27] place a message-passing kernel on each node, supporting processes and communication between them via explicit messages. This kernel supports both local communication—communication between processes on the same node—and nonlocal or remote communication (sometimes implemented via a distinguished network manager process).

Manuscript received April 21, 1989; revised March 17, 1990. This work was supported in part by the Defense Advanced Research Projects Agency, the National Science Foundation, and International Business Machines. An earlier version was presented at the 14th International Symposium on Computer Architecture.

U. Ramachandran is with the School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332.

M. Solomon and M. K. Vernon are with the Department of Computer Science, University of Wisconsin-Madison, Madison, WI 53706.

IEEE Log Number 9036217.

Access to system services is requested via protected procedure calls in a traditional system, whereas in a message-based operating system it is requested via message passing. While a simple procedure call costs just a few instructions, and a protected procedure call (kernel call) costs a few hundred instructions, IPC costs a few thousand instructions in several systems that we studied [20]. Since message exchange is the basic kernel mechanism in message-based operating systems, the performance of the system depends crucially on the rate of message exchange. Our measurements (see Section II and [24]) as well as the measurements of others [6], [11] indicate that for small messages (which make up the vast majority of all messages sent [6]), the limiting factor is the high processing overhead that is incurred in message passing rather than limited network bandwidth or the time to copy messages from buffer to buffer.

There are two important figures of merit in this environment: roundtrip time, and message throughput. Roundtrip time is the elapsed time seen by an application between sending a message and receiving a reply from the intended receiving process. This figure of merit affects an individual application's performance. Message throughput is a global figure of merit that determines the performance of the entire system. Informally, it is the number of messages that the system handles per unit time. We show that a modest amount of additional hardware can significantly improve message throughput and average roundtrip time in a multiprogramming environment. We also show that additional hardware support in the form of high-level bus primitives affords even greater improvement in communication subsystem performance.

A. Background and Related Work

Available hardware support [1], [14] and previous modeling studies [12], [29] address the issue of off-loading communication protocols onto front-end processors, and provide evidence that this approach can have a significant performance payoff. However, these and other previous proposals of hardware support for interprocess communication (see survey in [21]) are more limited than the study reported in this paper in several respects.

First, previous work generally assumes "communication" to be the work that is performed by the operating system to satisfy nonlocal requests. However, for message-based operating systems, measurements by us (see Section II and [24]) and others [6] show that there is a high processing overhead for local communication as well. Existing hardware support for interprocess communication takes the form of

1) operations on structured data types in the instruction set architecture of the processor (often through microcode) such as

...
Selected Publications in Full
Synchronization with Multiprocessor Caches *

Joonwon Lee       Umakishore Ramachandran

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332 USA

e-mail: joon@sybil.gatech.edu, rama@gatech.edu

Abstract

Introducing private caches in bus-based shared memory multiprocessors leads to the cache consistency problem since there may be multiple copies of shared data. However, the ability to snoop on the bus coupled with the fast broadcast capability allows the design of special hardware support for synchronization. We present a new lock-based cache scheme which incorporates synchronization into the cache coherency mechanism. With this scheme high-level synchronization primitives as well as low-level ones can be implemented without excessive overhead. Cost functions for well-known synchronization methods are derived for invalidation schemes, write update schemes, and our lock-based scheme. To accurately predict the performance implications of the new scheme, a new simulation model is developed embodying a widely accepted paradigm of parallel programming. It is shown that our lock-based protocol outperforms existing cache protocols.

1 Introduction

Cache memories have been used to reduce memory access latency in uniprocessors. In bus-based shared memory multiprocessors they may additionally reduce bus contention. However, private caches introduce the cache coherence problem [8]. The shared bus enables each cache controller to monitor the bus traffic and initiate appropriate actions to keep the shared data coherent. A group of cache coherence schemes called snooping cache protocols use this feature [13, 16, 18, 19, 24]. They are implemented in hardware and may not be visible to the programmer.

Most snooping cache protocols (Section 2) do not take into account synchronization requests that usually precede shared data accesses. Therefore, strong coherence among multiple copies is blindly enforced resulting in possibly unnecessary bus traffic for invalidation and data transfer. Usually, access to shared data is acquired via synchronization methods such as locks, semaphores, and barriers. Thus there is additional delay in accessing the synchronization variables and then acquiring the actual data.

In this paper, we present a cache coherence protocol supporting lock primitives (Section 3). Our scheme utilizes locking information provided by the software (e.g., compiler) to distinguish between shared-locks and exclusive-locks. We construct a distributed hardware-assisted FIFO queue of processors waiting for a given lock. With this scheme, the cache mechanism emerges as a visible part of the architecture since programmers should understand it to develop efficient parallel programs.

Efficient interprocessor synchronization and mutual exclusion are imperative to assure good performance for parallel programs. Therefore, we must evaluate the synchronization efficiency of cache protocols (Section 4).

Evaluating a multiprocessor system is a challenging task. Trace driven simulation has been used for multiprocessor evaluation [1, 11, 12, 25], but tracing parallel programs has so far been restricted to a small number of processors. Furthermore, the trace is affected by the host multiprocessor’s architectural characteristics such as cache protocol, synchronization primitives, and interconnection network. As Bitar [4] points out using traces generated by software makes it difficult to verify the validity of the predicted results. Therefore, we have developed a new method for the evaluation of cache protocols. Our simulation model and some results are presented in Section 5.

2 Snooping Caches

Hardware cache coherency schemes for shared-bus multiprocessors have evolved into two categories, namely, invalidation schemes and write update schemes.

In invalidation schemes [13, 16], a write to a cached line results in invalidating copies of this line present in other caches. If writes to cached data always trigger invalidations, the bus is easily saturated with even a few participating processors [27]. To reduce the invalidation rate, Goodman developed the write once protocol [13] in which only the first write to a cached data updates main memory, and is used as a cue by other caches for invalidating their own copies. The Berkeley protocol [16] assumes an invalidation line on the bus to explicitly invalidate peer caches. Since an invalidation is induced even with a first
write to data that is not being shared, there still exist redundant invalidations. These schemes lead to repeated invalidations and frequent data transfers when data is actively shared by many processors.

In write update schemes [18, 24], writes are broadcast and caches with matching entries update their corresponding cache lines. Using a probabilistic simulation model, Archibald and Baer [2] have shown that write update schemes outperform invalidation schemes. However, using trace driven simulation, Eggers and Katz [11] have shown that neither scheme dominates entirely. The performance of both schemes is sensitive to the sharing pattern. Particularly important is the length of the uninterrupted sequence of write requests interspersed with reads to a shared cache line by one processor (referred to as a write-run in [11]). The length of a write-run is the number of writes in that write-run. A write-run is terminated when another processor reads or writes the same cache line. Every new write-run requires an invalidation and data transfer. When write-runs are short the invalidation scheme generates frequent invalidations and the write update scheme generates equally frequent updates. Since the total cost for invalidation plus data transfer is much higher than the update of one word, invalidation schemes are inferior for this sharing pattern. For long write-runs, write update schemes perform poorly in comparison to invalidation schemes because of frequent write broadcasts. Repeated updates are clearly redundant given long write-runs. The optimal strategy would be to drop cache lines from all processors other than the writing processor. Thus the performance of both schemes critically depends on the sharing pattern.

In [15], a dynamic cache scheme called competitive snoopy caching is introduced. Among several variants presented in the paper, "Limited Block Snoopy Caching" is the most practical. First, the protocol begins by operating like a write update scheme. But, when the length of the write-run exceeds a threshold the cache scheme switches to the invalidation strategy. An important feature of this protocol is read snooping which allows caches to grab a cache line that is being transferred on the bus. This feature allows several readers to acquire a cache line in a single bus transaction following a long write-run. This cache scheme guarantees a lower bound for any sequence of memory requests when the threshold is equal to the block size.

Let \( p \) be the block size, which is also the threshold. Let \( n \) be the number of writes to a single shared address, \( l \) be the average write-motion distance, \( k \) be the number of write-runs \( l = n/k \). For simplicity, let \( k \) be a multiple of \( p \). Since there is a validation scheme costs \( k \times C_{\text{X}} \) for invalidating and transferring a block. The subscripts of \( C \) denote the sharing pattern. For validation, \( X \) for block transfer. Note that \( C_{\text{T}} \) is usually much smaller than \( C_{\text{X}} \) for invalidation scheme. The optimal strategy would be to drop cache lines. Since \( C_{\text{X}} \) is roughly \( p \) times large, the invalidation scheme generates much more bus traffic for cache coherence.

| Table 1: Cost for cache coherency. |
|-------------------|-------------------|-------------------|
| \( \text{Block Size} \) | \( \text{Cost} \) | \( \text{Snoopy} \) |
| 8 | 2.67 | 1.0 |
| 9 | 1.0 | 1.0 |
| 10 | 0.89 | 1.0 |
| 20 | 0.4 | 1.0 |

Thus in the worst case, the competitive snoopy scheme is

\[
\text{cost} = nC_{\text{W}} + aC_{\text{X}} = nC_{\text{W}} + apC_{\text{W}} \leq 2nC_{\text{W}}
\]

The protocols considered thus far treat cache coherence problems in isolation and do not consider synchronization issues. Bitar and Despain [5] propose a scheme in the cache accepts lock and unlock commands from a processor in addition to the traditional read and write requests. On receiving a lock request, the cache adds the address of the line in a specific register called the busy-wait register and the cache holding the lock sets its cache line state to lock-wait indicating there is at least one waiting processor. Upon receiving a unlock request, a cache changes its cache line state to invalid and broadcasts unlock if its state was lock-wait. This lock scheme combines lock-based synchronization with the line transfer, thus performing locking in zero time. This scheme does not distinguish between read lock and write lock requests. Since reads are a large portion of shared data access, this scheme limits potential concurrency.

In [14], Goodman et al. suggest a synchronization primitive, QOSB, which can be used by the programmer for high-level synchronization operations. They present an efficient implementation of this primitive by exploiting the hardware cache consistency protocol of the multiprocessor. The first lock requester becomes the owner of the
lock. The next requester allocates a new cache line (the shared line) and the address of the requesting processor is stored in main memory as the head of a queue of lock requesters. Since the data field of the shared line is invalid, this space can be used to store the address of the next requester. To know when to respond to a new requester, a tail queue pointer is also needed. Therefore, whenever a new requester appears, memory accesses are required to maintain these two queue pointers. These memory accesses can be eliminated if the primitive is implemented for a single bus multiprocessor. Like Bitar and Despain, Goodman only implements exclusive locks.

3 Lock Based Cache Protocol

At about the same time the QOSB primitive was developed, we designed a lock-based cache protocol [22] supporting exclusive and non-exclusive locks. Our lock based protocol (LBP) improves on the schemes discussed above. We chose lock primitives because of their generality. Like the queueing mechanism used in QOSB, a distributed queue is constructed using participating cache lines. In our scheme, a sharable lock is distinguished from an exclusive lock. Sharable locks enhance concurrency and are needed to efficiently implement other synchronization operations.

The processor and the cache together form a node of the shared memory multiprocessor. Each node is assigned a unique id which we refer to in this paper as node-id. The handshake between the cache and the bus is explained as follows. The cache entertains six requests from the processor: read, write, read-lock, write-lock, read-unlock, and write-unlock. Read and write are deemed as accesses to non-shared data and the cache processes them as would a uniprocessor cache. The granularity of a lock is a cache line. Processors must wait until the current request is satisfied before generating new requests.

Consider the sequence of lock requests, P1:read-lock, P2:read-lock, P3:write-lock, P4:read-lock, P5:read-lock, P6:write-lock, as shown in Figure 1. The first requester (P1) obtains a read-lock, and the following requester (P2) shares the lock since the lock type is read. P3 waits for the lock because its lock type is write. P4 and P5 wait after P3 to ensure fairness, even though the current lock held by P1 and P2 is sharable. A peer-group is a group of read-lock requesters who concurrently share a lock {{P1,P2} and {P4, P5} are peer-groups). To implement a queue, each directory entry of a cache line has a next-node field containing the node-id of the next waiting cache if any. When a lock is released, the cache sends a wake signal to the next waiting cache (if any). Caches with waiting states must monitor the signals on the bus for the line address and their node-id. Note that the protocol described here assumes a single process per processor. In [22], we describe the multiprocessing case and discuss implementation issues on standard buses.

The possible states of a cache line are summarized in Table 2, where R, W are used to specify the lock type, T to signify the tail of the queue, V to indicate waiting state, and O for the ownership. State transitions are triggered by processor requests and/or bus activities. Note that the cache controllers only respond to lock and unlock requests on the bus since simple reads and writes are deemed to be for private lines. Therefore, the states in Table 1 apply only for shared lines obtained through lock requests. In the discussion to follow, we use lock and line interchangeably since lock acquisition is merged with the cache line transfer.

An owner cache has the latest copy of the line, so it provides the line to the other caches when requested. The line is written back to memory when a write-lock owner releases the lock. There is at most one owner of a lock even when the lock is shared. A lock state with a T suffix denotes that the cache is at the tail of the waiting queue and should respond to subsequent requests for that lock. Only the first requester within a peer-group can be a tail or an owner. A shared lock is released when the size of the peer-group reaches zero, so caches with read-lock ownership or awaiting ownership keep the size of the peer-group in a count field. The ownership persists even after the line is unlocked at the owner cache. Assigning read-lock ownership to the first requester may result in unnecessary cache entries since it is likely that the read-lock owner will be the first to release the lock. However, the alternative choice of giving ownership to the last one in a peer-group could generate more bus traffic to transfer the count variable to the new owner. The width of the count field is determined by the number of nodes in the system. Alternatively, the width may be determined by the maximum

<table>
<thead>
<tr>
<th>States</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALID</td>
<td>The line is invalid</td>
</tr>
<tr>
<td>WO</td>
<td>Write lock owner</td>
</tr>
<tr>
<td>WOT</td>
<td>WO at the tail</td>
</tr>
<tr>
<td>WOV</td>
<td>Waiting for a write lock</td>
</tr>
<tr>
<td>WOV T</td>
<td>WOV at the tail</td>
</tr>
<tr>
<td>R</td>
<td>Read lock holder</td>
</tr>
<tr>
<td>RV</td>
<td>Waiting for a read lock</td>
</tr>
<tr>
<td>RO</td>
<td>Read lock owner</td>
</tr>
<tr>
<td>ROT</td>
<td>RO at the tail</td>
</tr>
<tr>
<td>ROV</td>
<td>Waiting for a read lock owner</td>
</tr>
<tr>
<td>ROVT</td>
<td>ROV at the tail</td>
</tr>
<tr>
<td>O</td>
<td>Unlocked, but still an owner</td>
</tr>
<tr>
<td>OT</td>
<td>O at the tail</td>
</tr>
</tbody>
</table>

Table 2: States of a cache line
membership we would like to allow in a peer group. Each cache line has a directory entry (tag) with the fields as shown in Figure 2.

Figure 3 illustrates the state transitions. When a processor requests a read-lock, the cache broadcasts it on the bus resulting in one of following responses:

- The block came from the main memory (denoted as \( \text{hit}(M) \) in the state transition diagram). It means that the line is not locked by any cache. The memory system provides the line, and the cache changes the state to ROT since it is the first requester.

- The block came from another cache (denoted as \( \text{hit} \) in the state transition diagram). The current read-lock owner sends the cache line allowing the requester to share the lock. The receiving cache changes its state to R.

- A \( \text{wait} \) signal is detected on the bus. A peer cache in state ROVT sends this signal with its own node-id. Since the \( \text{wake} \) signal (to be discussed shortly) is addressed to the first requester in a peer-group, this node-id is necessary for the waiting nodes to receive the signal correctly. The receiving cache stores this node-id in the next-node field, and changes its state to RV.

- A \( \text{wait}(T) \) signal is detected on the bus. The signal comes from a cache in state WOVT or WOT, and signifies that the tail state is transferred to the requester. Therefore, the receiving cache changes its state to ROVT.

When a cache receives \( \text{read-unlock} \) from the processor, the state of the line is one of R, ROT, or RO. A cache line in the R state is simply changed to the state INVALID, and a read-unlock signal is broadcast on the bus to inform the owner to decrease the count. If the state is ROT or RO, it is changed to OT or O respectively after decrementing the count. The cache is still the owner even after its own processor releases the lock and is responsible for sending a wake signal when the count goes to zero. Even though we assume a single process per processor, a processor may request a lock after releasing a lock, i.e., it may request a lock when the state of the line is OT or O. This case is not shown in Figure 3 since it is treated as a sub-case of multiple processes per processor. Another simple solution without increasing hardware complexity is to allow the processor to be an owner again. In this case, the fairness between processors cannot be guaranteed.

In case of a write lock, it is not necessary for the owner to keep the count since only one writer is allowed at a time. If a \( \text{wait}(T) \) signal is detected after broadcasting a write-lock, the state is changed from INVALID to WOVT. However, it ceases to be at the tail when any subsequent request for a lock is observed on the bus. On receiving an appropriate wake signal, the cache controller changes waiting states, WOV, WOT, to owner states, WOT, WO respectively, and allows the processor to use the line. It is not necessary to broadcast a write-unlock. On receiving a write-unlock request from the processor, the cache changes the state of the line to INVALID, sends a wake signal enclosing the cache line to the next requester (if any) as indicated by the next-node field, and writes the line back to memory.

Bus signals include: read-lock, write-lock, read-unlock, hit, \( \text{hit}(M) \), wait, \( \text{wait}(T) \), and \( \text{wake} \). The \( \text{wait} \) signal is sent from the tail cache to the lock requester when the lock is unavailable. The \( \text{wait}(T) \) signal additionally transfers the tail state to the requester. The wake signal is sent to notify that the lock is released to a cache whose node-id was stored in the next-node field of the tag entry for the line. Our scheme requires 13 states in the cache controller compared to 4 in most other protocols, and a larger cache tag memory to implement the distributed hardware queue.

<table>
<thead>
<tr>
<th>state</th>
<th>next-node</th>
<th>count</th>
</tr>
</thead>
</table>

Figure 2: Tag fields of a cache line

Figure 3: State Transitions
Cache line replacement becomes tricky for lock-based protocols including ours. If the line being replaced either owns a lock or is waiting for a lock then special attention is required. In the latter case, the best strategy would be to delete this node from the queue of waiting requesters for this lock. In a bus-based scheme such as ours this strategy can be fairly easily implemented. Prior to replacement, the cache controller broadcasts this event on the bus along with the address of the successor. The predecessor (cache controller) in the queue updates its cache line entry on receiving this event.

Replacing a cache line that owns a lock is a bit more complicated. The most simple and straightforward solution is to disallow replacement of a cache line that owns a lock. However, this solution may be feasible only if a fully associative or a set-associative caching strategy is in use. If a direct mapped caching scheme is used then it is necessary to modify the memory system to hold the lock status, tag and count fields of the cache line in addition to the data field. We believe this can be done with a little added complexity to the cache controller, minimal change to the memory system, and support from the compiler. The compiler can allocate additional memory space for every shared read/write data structure for holding the auxiliary information. For instance, if the data block is 4 words, one word may be reserved by the compiler for storing the auxiliary information. The memory system has a bit for every block that indicates whether the block is locked or not, which is returned with every memory request. When a cache line is replaced, the cache controller checks the status of the cache line. If it is locked, then the cache controller writes back the data field along with the auxiliary information to the memory system. When the block is then reloaded from the memory, the cache controller extracts the auxiliary information returned in the data field and stores it in the cache line. If a processor makes a lock request and the block comes from the memory, the cache controller checks the lock bit. If it is set then it is an indication that some other cache currently holds the lock. Therefore, the requesting cache has to retry the lock request at a later time.

The preceding description is only conceptual. In an actual implementation, it is not necessary that the lock bit in memory be out of band data. It is perfectly reasonable, and in fact practically feasible, to keep the lock bit in band. The compiler has to preallocate this space in the cache line and the cache controllers have to agree on the location of this lock bit in the cache line. Keeping the lock bit in band allows the use of conventional RAMs and eliminates the need for any special purpose memory system design.

In the next section, we see how our lock-based protocol aids synchronization and mutual exclusion in a parallel programming environment.

4 Synchronization Issues

Efficient synchronization is imperative for multiprocessors since parallel programs tend to generate repeated requests for mutual exclusion, barrier, and operations on shared data structures. The inefficiency caused by synchronization is twofold: wait times at synchronization points and the intrinsic overhead of the synchronization operations. Reducing waiting time is the province of the programmers. Reducing synchronization overhead is a task for the computer architect. Hardware support for synchronization comes in various forms such as a special-purpose coprocessor (e.g. Sequent SLIC [3]), a combining network [6, 10], and a special bus for interprocessor communication [26]. Recently, researchers have been interested in incorporating synchronization into snoopy cache schemes [5, 14, 22].

In the following subsections, we consider how efficiently various cache schemes (including our own) support synchronization. Three synchronization scenarios are considered. In the first (parallel lock) we assume n processors are simultaneously competing for the same lock. The second scenario (serial lock) assumes that locks are requested only serially. Finally we consider barrier notify. Similar to Goedman's QOSB primitive, a binary semaphore can be implemented very efficiently with our scheme while other schemes require mutual exclusion and queue construction.

4.1 Lock

If the machine supports an atomic test_and_set primitive then mutual exclusion can be implemented as in [20]. However, implementation of the test_and_set on traditional cache schemes can create additional penalties due to the ping-pong effect [9]. Since the 'set' part of the test-and-set primitive involves a write to a shared data, a spin-lock may cause each contending processor to invalidate (or update) other caches continuously. The following method avoids the ping-pong effect by busy-waiting on the cache memory without modification.

\[
\text{repeat} \\
\text{while(LOAD(lock_variable) = 1) do nothing;} \\
/*/ \text{spin without modification} */ \\
\text{until(test_and_set(lock_variable) = 0);} \\
\]

But, it still generates considerable bus traffic when a lock is released since all the waiting processors try to modify the cache line, thus invalidating (or updating) the corresponding cache line of the other caches.

Suppose n processors are competing for a lock at the same time (labeled 'parallel lock' in Table 3). After loading the lock variable, each processor executes test_and_set, thus generating n invalidations and n - 1 block transfers if the invalidation scheme is used. When the first lock holder releases the lock, it invalidates other copies of the lock variable. This invalidation is followed by (n - 1) block transfers. Now n - 1 processors are competing for the lock. So, total cost to service all the n processors with
The invalidation scheme is
\[ \sum_{i=1}^{n} (iC_x + iC_t + (i-1)C_x + C_t) \]
where the first term inside the summation is for loading the lock variable; the second and the third terms are the result of test_and_set executed simultaneously by \( i \) processors, and the last term is for unlocking. Simplifying the summation we get \( n^2C_x + \frac{n(n+1)}{2}C_t \). With the write update scheme, invalidation is replaced with write update. The cost in this case is given by
\[ \sum_{i=1}^{n} (iC_w + C_w) \]
where the first term is for test_and_set and the next one is for unlocking. Both of these schemes have a complexity of \( O(n^2) \). The constant factor of the invalidation scheme is larger than the write update scheme. The total cost with our scheme is \( 2nC_b \) since each processor issues lock and unlock commands on the bus. The read snarfing feature of the competitive snoopy caching allows an efficient implementation of the parallel lock. An unlock operation results in one invalidation followed by one line transfer to all the waiting requesters. The first cache which successfully performs test_and_set invalidates all the other caches. This invalidation triggers another line transfer for loading the lock variable into all the remaining caches. So, the total cost for this scheme is \( 2n(C_x + C_t) \).

The other extreme of lock competition is when all the \( n \) processors are serialized, i.e., only one processor requests the lock at one time. With the invalidation scheme this situation costs \( n(C_x + 2C_t) \) since each processor executes load, test_and_set, and unlock. For the write update scheme it costs \( 2nC_w \), and for the competitive snoopy caching it costs \( n(C_x + 2C_t) \). This case is termed as 'serial lock' in Table 3.

A lock request is with or without an argument. When a process needs mutual exclusion for a specific variable, the variable is specified as an argument. Even when there is not a specific variable to be locked, an argument is used if there are more than one critical sections in an epoch of parallel computation. A lock request with an argument needs twice the accesses to cache lines, one for the synchronization variable, and one for the actual data. The QOSB primitive and our cache scheme merge the two accesses into one for a cache line, and hence would show even more impressive performance than the other schemes for such requests. In Table 3 lock requests are assumed to be without arguments.

### Table 3: Overhead of Synchronization Primitives.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>LBP</th>
<th>W-Invalidate</th>
<th>W-Update</th>
<th>Competitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel lock</td>
<td>( 2nC_b )</td>
<td>( n^2C_x + \frac{n(n+1)}{2}C_t )</td>
<td>( \frac{n(n+1)}{2}C_w )</td>
<td>( 2n(C_x + C_t) )</td>
</tr>
<tr>
<td>serial lock</td>
<td>( 2nC_b )</td>
<td>( n(C_x + 2C_t) )</td>
<td>( 2nC_w )</td>
<td>( n(C_x + 2C_t) )</td>
</tr>
<tr>
<td>barrier notify</td>
<td>( C_b )</td>
<td>( C_t + (n-1)C_x )</td>
<td>( C_w )</td>
<td>( C_t + C_x )</td>
</tr>
</tbody>
</table>

---

**Figure 4:** Barrier implementation with snooping cache schemes

4.2 **Barrier**

Barrier requires all participating processors to synchronize at a certain point (the barrier). Traditional implementation of the barrier uses a counter which may become a hot-spot. In [28] a software combining method is suggested to eliminate the hot-spot where processors contend to increment the counter. This method has a nice scalability property (\( O(\log_2 n) \) overhead). But, considering the limit on the scalability of bus-based systems (certainly below 100), the benefit of dispersing the hot-spot is offset by the overhead for shared memory access with the software combining approach. In [23], Sohi et al. describe a restricted form of fetch-and-add, namely, fetch-and-increment that allows all the participating processors simultaneously.

---

1A specific memory location is contended for by many processors simultaneously.

2The electrical characteristics of the bus as well as the traffic, limits the number of processors which can be attached to a shared bus.
cessors to perform the operation in a single bus transaction. Brooks implemented the "Butterfly barrier" with busy-waiting locks [7]. This scheme requires $2 \log_2 n$ lock operations for each processor. A more realistic implementation of the barrier with bus-based snooping caches is given in Figure 4. Processors arriving before the last one keep reading their copies of flag variable in their respective cache memories. When the last arrival induces a write to the flag variable, other copies are invalidated or updated according to the cache coherency protocol being used. So, the cost for notifying other processors of the last arrival is $C_f + (n - 1)C_x$ for the invalidation scheme and $C_w$ for the write update scheme. With the read sneaking feature, the competitive snoopy caching executes barrier notification with a cost of $C_f + C_x$.

Barrier implementation with our cache scheme is given in Figure 5. In our scheme the 'wake' operation for a peer-group enables the notify phase to execute in one bus transaction. For the counting phase simple mutual exclusion is used since it is unlikely that the contention for the counting variable would cause too much overhead with an efficient lock operation. However, note that the degree of contention for the counting phase depends to a large extent on the type of work that has been farmed out to the processors before the barrier. If all the processors take roughly the same amount of time to complete their respective computations then it is likely that the contention will be high. Under such circumstances the scheme proposed by Sohi et al. [23] may prove to be quite efficient.

While synchronization issues are important, it is somewhat artificial to look at them in isolation from the point of view of system performance. To know the relative performance of cache protocols, we need to study the completion times of parallel programs with a workload comprised of private and shared data accesses interspersed with synchronization requests.

5 Performance Evaluation

Evaluating multiprocessors is difficult because of the interaction between the processors and the lack of a standard suite of benchmark applications. Recently real multiprocessor traces have become available. In [1] a tracing facility called ATUM2 is used for capturing the traces of parallel programs. However, it does not generalize for a variety of architectures due to its requirement of special hardware support and its limited scalability. Eggers and Katz [11, 12] measured multiprocessor cache protocols with traces of a set of parallel programs. Since the traces are generated on a per processor basis by software method, they are more scalable than the ATUM2. Traces gathered by software tools are prone to problems such as omission of operating system calls, oversimplification of the execution times of various instructions, and manifestation of architectural characteristics in the traces. Such problems may result in distorting the execution pattern of parallel algorithms. Note that such distortions may not necessarily lead to erroneous results in so far as measuring invalidation rate or performance related to the invalidation rate. However, as Bitar [4] points out, trace driven simulation may not be effective in capturing the high-level interactions between processors for hypothetical architectures.

We present a new simulation model for the measurement of multiprocessors. This model represents a dynamic scheduling paradigm believed to be the kernel of several parallel programs [21]. The basic granularity is a task. A large problem is divided into atomic tasks, and dependencies between tasks are checked. Tasks are inserted into a work queue of executable tasks honoring such dependencies, thus making the work queue non-FIFO in nature. Each processor takes a task from the queue and processes it. If a new task is generated as a result of the processing, it is inserted into the queue. All the processors execute the same code until the task queue is empty or a predefined finishing condition is met. If there is a need to synchronize all the processors at some point, then a barrier operation is used. Figure 6 shows the pseudo code that each processor executes.

To simulate the memory reference pattern of each processor during task execution, a probabilistic model similar to the one developed by Archibald and Baer [2] is incorporated into our model. Additional features in our model are synchronization primitives, differentiation of synchronization variables from other variables, and different evaluation metrics. Many parameters are fixed not only because their effects are well studied in [2] but also our primary concern was to measure the effect of various synchronization mechanisms on protocol performance. The values of the parameters used in the simulation are summarized in Table 4. The degree of sharing will be fairly low during the execution of a task than during queue operation. So, 0.03 and 0.5 are assumed for the degree of sharing for these two cases respectively. Real traces [11, 25] show that the read ratio ranges from 0.6 to 0.9 depending on the application of traced programs. In [2] simulation was done varying
Figure 6: Programming paradigm with dynamic scheduling

<table>
<thead>
<tr>
<th>Parameters</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ratio of shared accesses</td>
<td>0.03, 0.5</td>
</tr>
<tr>
<td>number of shared blocks</td>
<td>32</td>
</tr>
<tr>
<td>cache hit-ratio</td>
<td>0.95</td>
</tr>
<tr>
<td>read ratio</td>
<td>0.85</td>
</tr>
<tr>
<td>main memory cycle time</td>
<td>4 cache cycles</td>
</tr>
<tr>
<td>block size</td>
<td>4 words</td>
</tr>
<tr>
<td>cache size</td>
<td>1024 blocks</td>
</tr>
<tr>
<td>lock ratio</td>
<td>50%</td>
</tr>
</tbody>
</table>

* 0.03: task execution, 0.5: queue access

Table 4: Summary of parameters used in simulation

out barrier synchronization. BD denotes the Bitar and Despain's scheme, and LBP is our scheme. The unit metric is 10,000 cycles. The Berkeley protocol shows an anomalous loss of efficiency as the number of processors n increases beyond 8. This loss of efficiency happens because the bus is already saturated (the measured average length of the bus queue supports this argument) and thus, useful work is delayed by queue access activity of each processor. This effect is the multiprocessor equivalent of thrashing induced by the greedy scheduling discipline. All the other schemes also show slowdown in speed-up with more than 8 processors. With 8 processors LBP completes in 20480 cycles while BP, the second best one next to LBP, completes in 25149 cycles. This performance gap grows as n increases. The performance gap between BP and LBP is due to increased concurrency provided by read-locks of LBP during the task execute phase of the simulation model (Figure 6).

Figure 8 depicts the performance with a large grain size of 500. The steep loss of efficiency with Berkeley protocol disappears in this case because the long processing time outweighs the overhead of queue access. Though the grain size is multiplied by 5, the completion times for all the schemes do not scale down by quite as much, which is to be expected because there is a constant overhead for accessing the queue. When n is 8 the performance gap between LBP and BP is 14303 cycles, which is more than three times the gap when the grain size is fine to medium.

The effect of barrier synchronization is shown in Figures 9 and 10. Irrespective of the protocol, the net effect of the barrier is to synchronize the queue access of all the processors thus aggravating the contention for this shared resource (see Figure 6). Hence for a given grain size, the inclusion of the barrier results in longer completion times for all the protocols (compare Figures 7 and 9, and Figures 8 and 10). With respect to Figures 8 and 10, Dragon protocol outperforms Berkeley protocol by larger gaps with barrier than when there is no barrier. The same is true in Figures 7 and 9, except for the anomalous behavior of Berkeley protocol beyond 8 processors in Figure 7. The reason for these gaps is evident from the cost functions developed in Section 4.1: Even though the barrier itself does not lead to significant performance gap, simultaneous lock requests (see 'parallel lock' in Table 3) after the barrier entails considerable expense for the Berkeley protocol as compared to the Dragon protocol, especially with large n. With n = 8 and fine to medium grain parallelism, Dragon is better than Berkeley by 8% when barrier is not used, and by 32% when barrier is used. The case when the grain size is large (500) shows a similar trend. LBP outperforms BD by a larger margin with barrier synchronization. When the grain size is fine to medium (100), LBP is better than BD by 4669 cycles if barrier is not required, and by 9249 cycles with barrier. The efficiency of LBP for barrier operation comes from the sharable lock that enables the notification to be done in one bus transaction.
Figure 7: Performance of cache schemes (grain size = 100, without barrier)

Figure 8: Performance of cache schemes (grain size = 500, without barrier)

Figure 9: Performance of cache schemes (grain size = 100, with barrier)

Figure 10: Performance of cache schemes (grain size = 500, with barrier)
6 Discussion and Conclusions

We presented a new lock-based scheme which incorporates cache coherency strategy with synchronization. Lock operations were used as the underlying primitives for cache coherency. As a waiting mechanism, a distributed queue was constructed in hardware using the cache line of each lock requester. Memory accesses for queue pointers were eliminated by storing the link information in the tag of the cache memory. The protocol distinguishes sharable lock from exclusive locks thus allowing increased concurrency for simultaneous readers. The invalidation, the write update, and the lock-based schemes were analyzed for synchronization operations to identify the source of inefficiency. For the evaluation of each cache scheme a new simulation model was developed which represents a widely used paradigm for parallel programming. The simulation results show that our scheme outperforms others by a considerable margin in the test cases.

There are several dimension in which our work can be extended in the future. The fetch-and-add [10] primitive is a powerful synchronization primitive and its utility has been demonstrated in combining multistage networks. It would be interesting to compare the performance and hardware complexity of the fetch-and-add primitive in a bus-based environment against our protocol. We showed earlier that read snarfing is a useful feature for synchronization. Detailed simulation study for examining the overall performance of read snarfing on snoopy cache protocols is another area of future research. VLSI implementation of cache schemes such as invalidate, update, Bitar and Despain, QOSB primitive, and ours is also being pursued to quantify the cost in terms of circuit complexity and understand the delay characteristics of the different cache protocols.

Acknowledgment

We thank Jim Goodman for his several constructive criticisms and suggestions. We thank our colleagues, H. Venkateswaran and Phil Hutto for reading earlier drafts of this paper.

References


Locks, Directories, and Weak Coherence - A Recipe for Scalable Shared Memory Multiprocessors*

Joonwon Lee    Umakishore Ramachandran

Technical Report GIT-CC-90/68
December 1990

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

joon@cc.gatech.edu, rama@cc.gatech.edu

Abstract

Bus based multiprocessors have the limitation that they do not scale well to large numbers of processors due to the bus becoming a bottleneck with the current bus technology. Lock-based protocols have been suggested as a possible way of mitigating this bottleneck for single bus systems with snooping ability. In this research, we are interested in extending lock-based protocols to general interconnection networks. Directory based cache coherence schemes have been proposed for such networks. We are investigating a combination of locking with directory based schemes. Further, most protocols in the literature until now, assume a strong coherence requirement. However, recent research has shown that it is possible to weaken this coherence requirement. Such an approach is expected to reduce the coherence overhead even further, making it an appealing one for building scalable systems.

*This work is supported in part by NSF grant MIP-8809268.
1 Introduction

In shared memory multiprocessors there is a need for defining a consistency model that specifies the order of execution of memory accesses from multiple processors. Lamport [Lam79] has proposed sequential consistency as the ordering constraint for correct execution of a multiprocess program: The multiprocessor execution of the program should have the same effect as a sequential execution of any interleaving of the operations of all the processes (that comprise the program). The allowed interleavings are those that preserve the program order of operations of each individual process.

With the sequential consistency model, read and write operations are sufficient to implement synchronization operations correctly. However, this model is inherently inefficient since it imposes a strong ordering constraint for all memory accesses regardless of the usage of shared data. Further, each memory access has to wait until the previous memory access is completed. Thus large scale shared memory multiprocessors are expected to incur long latencies for memory accesses if this ordering constraint is imposed, leading to poor performance. Further, such long latencies seriously hamper the scalability of shared memory multiprocessors.

In parallel program design, it is not unusual to use synchronization operations to enforce a specific ordering of shared memory accesses. Based on this observation Dubois et al. [DSB86] have proposed weak ordering that relaxes the ordering constraint of sequential consistency by distinguishing between accesses to synchronization variables and ordinary data. Their model requires (a) that synchronization variables be strongly consistent, (b) that all global data accesses be globally notified before synchronization operations, and (c) that all global data accesses subsequent to the synchronization operation be delayed until the operation is globally performed. Thus this model requires strong consistency of global data accesses with respect to synchronization variables.

A synchronization operation usually consists of acquire and release steps, e.g. lock and unlock, P and V of semaphores, barrier-request and barrier-notify. The acquire step (such as P or lock) need not be strongly consistent with respect to global memory accesses issued before it. Likewise, global memory accesses following the release step (such as V or unlock), need not wait until the synchronization operation is globally performed. This observation enables several extensions [AH90, CLLG89] to weak ordering, and thus provides more flexibility in machine design.

Private caches significantly reduce memory latencies and network contention in shared memory.
multiprocessors, provided an efficient cache coherence scheme is devised. For bus-based multiprocessors snooping cache protocols have been popular since they provide the sequential consistency model without much overhead. However, snooping caches rely on a fast broadcast capability not available in more scalable interconnections such as a multistage interconnection network. However, private caches are indispensable for reducing memory latencies, despite the fact that scalable interconnections complicate coherence maintenance. Because of their scalability, directory-based cache coherence protocols have been proposed for large scale shared memory multiprocessors [CG89, CFK90]. Until recently, such protocols implement the strong sequential consistency constraint used in the snoopy cache protocols. However, given that global operations such as invalidation and updates are expensive in scalable interconnects, it is important to incorporate weak consistency models in such protocols.

In this paper, we present a directory-based caching scheme based on a consistency model in which strong consistency is enforced only for cache lines accessed by lock operations. Other accesses are deemed to be for private data. This protocol is similar to write-broadcast protocol [TS87, McC84] with the difference that updates are only sent to processors that request them. Lock requestors wait in a FCFS queue organized with pointers in each cache line and the directory. In the next section we consider the effect of caching schemes on consistency models. The motivation for and applications of our new caching scheme are presented in Section 3. Section 4 describes our cache protocol and its implementation issues. Finally, the performance potential of our scheme is discussed in Section 5.

2 Cache Consistency

For the purposes of this paper the multiprocessor model is a uniform shared memory one: each processor has a private cache; an interconnection network connects the processors with the memory module(s); and a cache coherence scheme assures consistency using some consistency model. The sequential consistency model for cache coherence is the most straightforward one to comprehend from the point of view of programming. This model parallels the database transaction model of single copy serializability in the way it ensures correctness of concurrent execution of operations.

1 Similar ideas have been proposed in our previous work [LR90], and IEEE SCI protocol [IEE89]
Therefore, it is easy to see that this consistency model relieves the programmer from having to worry about the order of execution of individual operation in writing parallel programs. Thus, until recently multiprocessor cache coherence schemes have used this consistency model.

The implementation cost (in time, network traffic, and circuit complexity) of a cache protocol depends on the choice of interconnection network. When the interconnection network is a single shared bus it is relatively easy to implement the sequential consistency model for cache coherence since (a) each processor has the ability to observe the (read/write) events from all the other processors by snooping on the bus, and (b) the bus provides a fast broadcast capability. This observation has led to a group of cache coherence protocols often referred to as snooping cache protocols [Goo83, KEW+85, PP84, KMRS88, TS87, McC84, LR90] for bus-based systems. The snooping ability of the processors allows the directory information to be distributed among the private caches.

It is fairly well-known that single bus systems are incapable of scaling to very large numbers of processors due to the bus becoming a bottleneck beyond a few processors [AB86]. Therefore, there has been a resurgence of interest in developing efficient cache coherence schemes for more scalable interconnection networks [MHW87, GW88, Wil87, BW87]. While some of these have been extensions of single bus snooping cache protocols to multiple buses, some researchers have been investigating an alternative approach, namely, directory-based protocols. In this approach, the directory information is centralized at the memory module(s) instead of being distributed in the private caches, thus making it is possible to implement the sequential consistency model for non bus-based networks. However, the lack of a fast broadcast capability in such networks leads to an inefficient implementation of the sequential consistency model.

All of the protocols until recently have used the sequential consistency model for assuring correctness. However, recently researchers have observed that the sequential consistency model is too strong, and have developed weaker models of consistency [SD87, AH90, LS88, HA89, ABM89]. Researchers have shown that it is possible to achieve a more efficient implementation of multiprocessor cache protocols with weaker models of consistency [ABM89, AH90, CLLG89].
3 Consistency and Synchronization

The evolution of multiprocessor cache protocols has three distinct aspects: development of different models of consistency, study of trade-offs between centralized and distributed directories, and innovation in the choice of cache primitives. We discussed the first two aspects in the previous section. In this section we discuss the motivation for developing new cache primitives. Early cache protocols assumed the traditional uniprocessor interface between the processor and the cache, namely, the cache responds to read and write requests from the processor. However, in a multiprocessor data is potentially shared between processors. To account for this, the cache protocols extend the functionality of the private caches for read/write accesses from other processors via the network, satisfying some chosen model of consistency.

However, when there is sharing there is synchronization, that usually governs such sharing. By ignoring the synchronization aspect, most of the early protocols treated the coherence problem for shared data in isolation. In reality, synchronization and data coherence are intertwined. Realizing this, researchers have proposed primitives for synchronization via the caches [GW88, BD86, LR90].

The data accesses from a processor can be grouped into three categories: access to private data, access to synchronization variables, and access to shared data. If permission to access to shared data is acquired through some synchronization mechanism then strong coherence would be required only for the access to the synchronization variables. Synchronization operations may be used in a program for one of three purposes:

- Case 1 The operation guarantees the atomicity of a set of operations (transaction) on shared data. A lock is usually associated with the shared data for this purpose. The lock governs access to the shared data.

- Case 2 The operation provides mutual exclusion to a critical section in the program.

- Case 3 The operation signals the completion of an epoch of a computation to other processes (e.g. barrier synchronization).

In each case the synchronization operation itself requires strong coherence enforcement. If the cache is capable of recognizing a synchronization request distinct from other requests, then it is
relatively straightforward to enforce the strong coherence requirement: stall the processor until the synchronization operation is performed globally. However, once the synchronization operation is performed globally there is latitude in how other accesses are performed. For instance in case 1, if the hardware has knowledge of the shared data associated with each lock then it can "batch" the global propagation of all the updates to shared data that happened in this transaction upon release of the lock. Similarly, in case 2 the synchronization variable governing access to the critical section needs to be strongly coherent but accesses inside the critical section need not be globally notified until exit from the critical section. Case 3 is similar to case 2 in that accesses done during the epoch of the computation need not be globally propagated until the notification phase (similar to exiting a critical section) of the barrier operation. These arguments are along the lines of those put forward by Scheurich et al. [SD87], and Adve and Hill [AH90] for justifying weak ordering.

Thus if the hardware is capable of recognizing a set of synchronization operations, then subsequent data accesses can be treated as not requiring any consistency. Therefore, any cache protocol that performs consistency maintenance for all shared accesses is doing more work than it has to. This argument is the motivation for our cache coherence protocol that is described in the next section.

In the discussion thus far we have assumed that a process always acquires permission to access data or code through some synchronization mechanism. This need not always be true. There are applications in which it may be sufficient that a read of a shared location return the most recent value. Further it may not be necessary that the value be updated or invalidated if other processors write that location subsequently. An example is a monitoring process that wants to read the value just once. Parallel game programs are another example where it may be sufficient to perform the updates more lazily. Once again any such optimizations to reduce the network traffic requires that the hardware (cache controller) support additional primitives, and that the software be written to take advantage of such primitives. Needless to say such optimizations allow the scalability of such machines by reducing the network traffic.

Our proposed cache protocol (to be described next) has four features: (a) it provides hardware recognized locking primitives to handle all three cases discussed above; (b) it exploits the weak ordering principle by performing updates to shared locations more lazily; (c) it distinguishes be-
4 Protocol Description

In this section, we present a directory-based cache protocol based on the consistency requirements discussed in section 3. Our scheme assumes cooperation with the software in generating appropriate requests for the desired consistency.

The cache entertains seven commands from the processor: read, read-update, write, read-lock, write-lock, unlock, and flush. Read and write are regarded as requests for private data requiring no consistency maintenance. Read-update is similar to read except that it requests updates for the cache line. This is a dual to the write-update schemes [TS87, McC84] in that the updates are receiver initiated as opposed to sender initiated. Read-lock, write-lock, and unlock requests combine processor synchronization with data coherence. Read-lock provides the requesting processor with a non-exclusive copy of the cache line that is guaranteed by the protocol not to change. Similarly, an exclusive copy of the cache line is provided to the write-lock requester. The processors have to explicitly perform unlock to release the cache lines acquired under a lock. The flush operation purges a cache line updating memory words that are modified in the cache line. The main memory, in turn, sends updates of this line to other processors if need be.
Figure 1-a illustrates tag fields of a cache line. The *update* bit of a cache line indicates whether updates have been requested for this line. The next *k* bits denote the modified word(s) of the cache line respectively, where *k* is the number of words per cache line. Only the modified words are written back to memory when the cache line is replaced. The *lock-status* denotes if the cache line is locked, and if so, the kind (read or write) of lock. The next two fields are pointers which are used to construct a doubly-linked list for processors waiting for the same lock. If the lock-status is not locked and the *previous* pointer is not null, this cache line is waiting for a lock. If the *next* pointer is null, this processor is at the *tail* of the queue.

Figure 1-b shows tag fields of a directory entry. The first field, *update-list*, consists of *n* bits, one for each of the *n* processors in the system. Each processor has a designated bit in this field which is set when it makes a read-update request, and reset when the cache line is replaced by the processor. Upon write backs to main memory, this field is checked and the updated cache line is sent to processor *i* if the *i*-th bit of the update-list is set. The *queue-tail* field is a pointer to a processor which was the last lock requester for the memory block. If queue-tail is *null* this memory block is currently unlocked.

For a read request, if the requested data is in the cache, the cache provides the data regardless of the state of the cache line. If the data is not in the cache, the request is forwarded to the main memory. Regardless of the state of the corresponding directory entry, the main memory sends the data to the requesting cache. A read-update request can be serviced by the cache if the update bit of the cache line is set. Otherwise, this request is forwarded to the main memory. The main memory provides the requested data and sets the appropriate bit in the update-list. Once this bit is set, the processor will be supplied with the new data whenever the memory block is written back by another processor. The write backs occur when a cache line is replaced, unlocked, or flushed.

Upon a lock request it is sent to the main memory and the directory entry is investigated. If the memory block is not locked, the address of the requester, say *P_i*, is stored in the queue-tail field of the directory entry. The memory block is sent to the requester and the cache sets the lock status of the cache line according to the type of the lock. If the memory block is locked, the lock request is forwarded to the processor addressed by the current queue-tail field and *P_i* becomes the new queue-tail field of the directory entry. The *next* pointer of the current queue-tail processor is now
made to point to $P_i$. $P_i$'s lock request is granted if the cache line of the tail processor is currently holding a read-lock, and $P_i$'s request is also a read-lock. Figure 2 illustrates a queue for a series of lock requests, $P1$:read-lock, $P2$:read-lock, $P3$:write-lock. $P1$ and $P2$ currently hold read-locks while $P3$ is waiting for a write-lock. For more details on this lock-based protocol the reader is referred to a companion paper [LR90].

Upon an unlock request, the cache controller releases the lock to the next waiting processor (if any), and writes the cache line to the main memory (if necessary). When a write-lock is released there could be more than one processor waiting for a read-lock. The lock release notification goes down the linked list until it meets a write-lock requester (or end of the list), and thus, allows granting of multiple read-locks. When a processor unlocks a read-lock and the processor is not the sole lock owner, the list is fixed up similar to deleting a node from a doubly-linked list. When a lock is released by a tail processor, the main memory marks the tail pointer null. Note that the unlocking processor is allowed to continue its computation immediately, and does not have to wait for the unlock operation to be performed globally.

Replacing a cache line that is a part of the list of lock requesters is a bit more complicated. The most simple and straightforward solution is to disallow replacement of such a cache line. However, this solution may be feasible only if a fully associative or a set-associative caching strategy is in use. If a direct mapped caching scheme is in use it is necessary to modify the tag handling mechanism of cache controllers. When a locked cache line is replaced, the cache controller preserves the lock.

![Figure 2: A waiting queue: doubly linked list](image-url)
status and the two pointer fields in the tag memory of the cache line, i.e., only the data part and address tag part of the line is evicted. We assume that the compiler allocates at most one lock variable to a cache line. Since the new line to occupy this evicted cache line is for ordinary data, it does not interfere with the list structure for the lock (guaranteed by the compiler). Cache controllers do not perform address tag matching for link update operations to allow the operations to be performed correctly even after the locked cache line is replaced. A subsequent access (read, write or unlock) to the lock variable will reload the replaced cache line.

However, restricting one locking process per processor at a time may be too restrictive. Since a processor holds (or waits for) only a small number of locks at a time, a small separate fully-associative cache for lock variables can be an efficient method to eliminate the restriction. This cache has tag fields as shown in Figure 1. Other ordinary shared or private data are stored in another cache which does not have tag fields for locks such as lock-status, previous, and next.

Another hardware requirement for our protocol is a write buffer. The flush operation is supposed to write back all the updated cache lines. However, detecting updated cache lines cannot be done by a compiler statically since shared data may be referenced through pointers. Searching all the cache lines by hardware is very inefficient. So, our protocol needs a write buffer which holds updated data until they are globally performed.

5 Performance Analysis

In this section, we present a simple performance analysis of our cache protocol. We compare ours with a write-back invalidation full-map directory (WBI) protocol that uses the sequential consistency model. First, we analyze the performance of each scheme for lock and ordinary memory accesses. Based on this analysis, cache schemes are compared for common structures of parallel programs such as task queue model, and static scheduling model.

5.1 Lock Operations

First, we derive expressions for time and network message complexity for acquiring a mutually exclusive lock to enter a critical section for the WBI protocol. Recall that with the sequential consistency model, read and write operations are sufficient to implement synchronization operations
correctly. An implementation of synchronization operations can be more efficient with an atomic test-set primitive which most architectures provide. However, implementation of the test-set on traditional cache schemes can create additional penalties due to the ping-pong effect [DSB88]. Since the ‘set’ part of the test-set primitive involves a write to a piece of shared data, a spin-lock may cause each contending processor to invalidate (or update) other caches continually with the sequential consistency model. The following test-test-set method avoids the ping-pong effect by busy-waiting on the cache memory without modification.

\[
\text{repeat} \\
\quad \text{while(LOAD(lock_variable) = 1) do nothing;} \\
\quad /* spin without modification */ \\
\quad \text{until(test-set(lock_variable) = 0);}
\]

But, this method still generates considerable network traffic when a lock is released since all the waiting processors try to modify the cache line, thus invalidating (or updating) the corresponding cache line of the other caches.

Figure 3 shows a timing diagram for the case when \( n \) processors execute test-test-set at the same time. The lock variable is assumed to be initially in the dirty state in a peer cache. \( t_{nw} \) denotes the message transit time on the network. For simplicity of analysis, we assume the same time for all types of network messages: request to the main memory, query from the main memory to a cache; acknowledgment, and block transfer. Let \( t_m \) be the memory access time; \( t_D \) and \( t_C \) denote directory checking time of the main memory and the cache memory, respectively. In this analysis, we assume requests to memory are queued by order of arrival if the memory is busy. Figure 3-a shows the parallel execution of load instruction of test-test-set by \( n \) processors. \( n \) read requests are issued at the same time \( (t_{nw}) \). For the first request (top time line in Figure 3-a), the main memory performs a directory checking \( (t_D) \) and sends a request for the block to the cache which has the dirty copy \( (t_{nw}) \). After checking the cache directory \( (t_C) \), the cache controller sends the block to the main memory \( (t_{nw}) \). Once the requested data arrives it is written to the main memory \( (t_m) \), and sent to the requester \( (t_{nw}) \). After these steps the main memory has a valid copy. So, subsequent read requests require just directory checking \( (t_D) \), wait for memory \( (t_{wait}) \), memory read \( (t_m) \), and block transfer \( (t_{nw}) \). Note that memory operations cannot be overlapped.
Assuming that the lock-variable is 0, all the processors are going to execute test-set operations next. But these test-set requests can be serviced only after the last load request finishes. These requests can be serviced by the memory only after the last \( t_m \) request of Figure 3-a finishes. Referring to the top line in Figure 3-b, the first test-set operation \( (t_{nw}) \) invalidates valid copies which were acquired by the above load operations by \( n - 1 \) processors. Sending \( n - 1 \) invalidation messages is a sequential operation for the main memory \( ((n - 1)t_D + t_{nw}) \). After invalidating local copies \( (t_C) \), all these \( n - 1 \) cache controllers send acknowledgments to the main memory. Receiving these acknowledgments is also a sequential operation \( ((n - 1)t_D) \). After that, memory acknowledges the test-set of the first processor \( (t_{nw}) \), allowing it to complete successfully. For the next test-set operation (second time line in Figure 3-b) the main memory requests the block from the cache that has a dirty copy after checking its directory \( (t_D + t_{nw}) \). The block is transferred to the main memory from the cache \( (t_C + t_{nw}) \), and then to the test-set requester \( (t_D + t_{nw}) \).
Among the \( n \) processors executing the test\&set operation, only one completes it successfully. The remaining \( n-1 \) processors re-execute the load operation and busy-wait on cached copies of the lock variable until the lock variable is reset by the lock owner. Since all the waiting processors have valid copies (in their caches), the reset operation invalidates \( n-1 \) copies. Now, \( n-1 \) processors compete for the lock simultaneously with one dirty copy in the resetting processor’s cache, which is the initial condition that was assumed when \( n \) processors competed for the lock. This parallel lock contention is repeated \( n \) times, with one less contender in every round, until all the requesters get serviced.

Since the performance of large-scale multiprocessors depends on the amount of traffic on the interconnection network, the number of messages generated by a cache protocol can be used as a metric for evaluating the performance of cache protocols. Let \( f_m(n) \) be the number of messages generated for granting a lock to one processor when \( n \) processors are competing simultaneously. Then \( f_m(n) \) is computed as the sum of (refer to Figure 3): 4+2\((n-1)\) for load operations, \( 2+2(n-1) \) for the first test\&set, \( 4(n-1) \) for the next \( n-1 \) test\&sets, \( 4+2(n-2) \) for \( n-1 \) re-loads, and \( 2+2(n-1) \) for reset. Therefore, \( f_m(n) = 12n - 2 \), and the total number of messages for servicing \( n \) lock requests is

\[
\sum_{i=1}^{n} f_m(i) = 6n^2 + 4n
\]

For measuring the execution time for the \( n \) parallel lock requests, we made the following assumptions: a) Lock contention of other processors do not delay the memory accesses of a processor holding the lock. b) The sojourn time inside the critical section is long enough so that the reset operation comes after all the test\&sets and re-loads are completed. c) The network transit time is a constant independent of switch contentions. From Figure 3-a, load operations take \( 3t_{nw} + t_C + t_D + t_m + (n - 1)(t_D + t_m) \) until the last memory operation completes. The first test\&set takes \( 2(n - 1)t_D + 3t_{nw} + t_C \) before successfully entering the critical section. Let \( t_{cs} \) be the processing time in the critical section, and \( t_1 \) be the time at which the first lock holder completes unlock operation. Then \( t_1 \) can be expressed as a function of \( n \),

\[
t_1 = f_t(n) = 3t_{nw} + t_C + t_D + t_m + (n - 1)(t_D + t_m) + 2(n - 1)t_D + 3t_{nw} + t_C + \ldots
\]
The first line is for \( n \) loads, the second line is for test&set, the third line is for a critical section, and the fourth line is for unlock. Since \( t_i = t_{i-1} + f_i(n - i + 1) \) for \( 2 \leq i \leq n \), the completion time for \( n \) processors executing the critical section is

\[
t_n = \sum_{i=1}^{n} t_i = nt_{cs} + 10nt_{nw} + 3nt_C + \frac{n(n+1)}{2}t_m + \frac{(5n-4)(5n-3)}{2}t_D
\]

where \( t_{cs} \) is the critical section time, \( t_{nw} \) is the network message time, \( t_D \) is the directory check time, \( t_m \) is the memory read time, and \( t_C \) is the test&set time.

For our scheme, the timing diagram for acquiring a mutually exclusive lock (write-lock) is shown in Figure 4. The first lock operation (top time line of Figure 4-a) requires one network message for lock request \( (t_{nw}) \), a directory check at the memory \( (t_D) \), memory read \( (t_m) \), and a network message for granting the lock \( (t_{nw}) \). For each subsequent request, the memory does a directory check \( (t_D) \), sends a message to the current tail \( (t_{nw}) \), which updates its directory \( (t_C) \), sends an acknowledgment \( (t_{nw}) \), which is received by the memory \( (t_D) \), and results in a message from the memory to the new tail \( (t_{nw}) \). For \( n \) lock operations, \( 2+4(n-1) \) messages are issued. Each unlock
operation generates 2 messages (See Figure 4, one for unlock, and one for waking up the next requester). So, for servicing $n$ lock requests, total number of messages generated is

$$2 + 4(n - 1) + 2n - 1 = 6n - 3$$

And, for the completion time,

$$t_1 = 2t_{nu} + t_D + t_m + t_{cs} + 2t_{nu} + t_D$$

$$t_i = t_{i-1} + t_{cs} + 2t_{nu} + t_D \quad (2 \leq i \leq n)$$

Therefore,

$$t_n = t_1 + (n - 1)(t_{cs} + 2t_{nu} + t_D) - t_{nu}$$

$$= nt_{cs} + (2n + 1)t_{nu} + (n + 1)t_D + t_m$$

The subtraction of $t_{nu}$ in the above equation accounts for the last unlock operation in which the main memory does not send a wake-up message.

The other extreme of lock competition to the parallel lock is when all the $n$ processors are serialized, i.e., only one processor requests the lock at a time. For the WBI protocol, each lock requester generates 4 messages for load, 4 messages for test&set, and 0 message for reset since the cache has a dirty copy of the lock variable. Thus, $n$ processors generate $8n$ messages. The time spent by each processor is $4t_{nu} + t_D + t_C + t_m$ for loading, $4t_{nu} + t_C + 2t_D$ for test&set, and $t_{cs}$ for the critical section. Since lock requests are occurring serially, the time for executing the critical section with the WBI protocol for each processor is $8t_{nu} + t_{cs} + 2t_C + 3t_D$. With our scheme, the time for each processor to execute the critical section serially is $3t_{nu} + t_D + t_{cs}$; with 2 messages for acquiring the lock and one message for releasing the lock.

### 5.2 Barrier Synchronization

Barrier requires all participating processors to synchronize at a certain point (the barrier). Since the barrier counter can be a hot-spot several approaches [YTL87, Bro86] have been proposed for distributing the contention. However, we assume a traditional implementation using a counter with lock operations on the counter for the performance analysis. The cost for a barrier-request is the cost
for lock request plus the cost for accessing the counter. So, the performance of the barrier depends on the arrival pattern. For the WBI protocol, the barrier-notify step causes \( n - 1 \) simultaneous read requests. If we assume serial arrivals for a barrier-request, each processor generates 8 messages for a serial lock, 8 messages for reading and writing the counter variable, 2 messages for reading a flag variable. The time taken for a barrier-request is \( 8t_{nw} + 3t_D + 2t_C \) for a lock, \( 8t_{nw} + 3t_D + 2t_C \) for incrementing the counter, and \( 2t_{nw} + t_D \) for reading the flag variable, where the time for incrementing the counter and the time for checking conditions are not included. So, each barrier request takes \( 18t_{nw} + 7t_D + 5t_C \) time, and generates 18 messages.

The barrier-notify is a write to the flag variable on which all the waiting processors busy-wait. This write causes invalidations of all the \( n - 1 \) copies. So, messages generated by a barrier-notify are 1 write request, \( n - 1 \) invalidation signals to caches, \( n - 1 \) acknowledgments, 1 acknowledgment to the writer, \( n - 1 \) re-load requests, and \( n - 1 \) block transfers. So, a barrier-notify generates a total of \( 5n - 3 \) messages. The time spent for a barrier-notify is \( t_{nw} \) for a write request, \( (n - 1)t_D + t_{nw} \) for sending \( n - 1 \) invalidations, \( t_C \) for invalidation at the caches, \( (n - 1)t_D + t_{nw} \) for receiving \( n - 1 \) acknowledgments, and \( t_{nw} \) for sending an acknowledgment to the barrier-notifier, resulting in a total time of \( 4t_{nw} + 2(n - 1)t_D + t_C \).

With our scheme, the barrier can be implemented by lock and read-update primitives. The counter is secured by lock operations and the busy-waiting is implemented by the read-update primitive. Processors arriving at the barrier increment the counter. If it is less than \( n \), the processor issues a read-update for the flag variable. The processor that arrives last at the barrier sets the flag variable and notifies this update using the flush primitive. A barrier-request generates one message for write-lock and one for unlock. Both write-lock and unlock each take \( t_{nw} + t_m \) time. When the flag is written and flushed by a processor, \( n - 1 \) processors waiting for updates will receive the update. So, it generates 1 flush request and \( n - 1 \) word transfers, and takes \( t_{nw} + (n - 1)t_D + t_{nw} \) time.

Summary of the costs for each synchronization scenario with the WBI protocol and ours is presented in Table 1 and 2.
<table>
<thead>
<tr>
<th>Synchronization Operation</th>
<th>WBI Protocol</th>
<th>Our Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Lock</td>
<td>$6n^2 + 4n$</td>
<td>$6n - 3$</td>
</tr>
<tr>
<td>Serial Lock</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Barrier Request</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>Barrier Notify</td>
<td>$5n - 3$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

Table 1: Number of messages generated by synchronization operations.

<table>
<thead>
<tr>
<th>Synchronization Operation</th>
<th>WBI Protocol</th>
<th>Our Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Lock</td>
<td>$nt_{cs} + 10nt_{nw} + n(n + 1)/2t_m + 5n(5n - 1)/2t_D$</td>
<td>$nt_{cs} + (2n + 1)t_{nw} + (n + 1)t_D + t_m$</td>
</tr>
<tr>
<td>Serial Lock</td>
<td>$8t_{nw} + 5t_D + t_m + t_{cs}$</td>
<td>$3t_{nw} + t_D + t_{cs}$</td>
</tr>
<tr>
<td>Barrier Request</td>
<td>$18t_{nw} + 12t_D$</td>
<td>$2(t_{nw} + t_m)$</td>
</tr>
<tr>
<td>Barrier Notify</td>
<td>$4t_{nw} + (2n - 1)t_D$</td>
<td>$2t_{nw} + (n - 1)t_D$</td>
</tr>
</tbody>
</table>

Table 2: Time taken by synchronization operations. $t_C$ is replaced by $t_D$ for simplicity. Costs for serial lock and barrier request are for one processor.

5.3 Shared Variables

While the performance of the invalidation-based directory scheme is sensitive to the sharing pattern of memory accesses, our scheme is independent of the sharing pattern since strong coherency is enforced only for lock requests. In this subsection, we analyze the time taken for accessing private/shared data during an epoch of computation for the WBI protocol. The analysis is based on two basic assumptions: the mean time between shared accesses is exponentially distributed, and shared accesses are uniformly distributed over $k$ shared variables. Following are the parameters used:

- $N$: total number of memory accesses
- $k$: the number of shared variables in words
- $h$: cache-hit ratio
- $sh$: ratio of shared accesses
- $r$: ratio of read accesses
\( w \): ratio of write accesses, i.e., \( 1 - r \)

\( t \): mean time between memory requests

The total time for accessing data can be divided into two components: private accesses and shared accesses. For both types of accesses, we assume the same hit-ratio. However, the cache-hit ratio may be decreased for shared lines by interferences from other processors such as invalidations and external reads of a dirty cache line. So, we develop the following equations:

\[
\begin{align*}
\text{total } \text{time} &= N((1 - sh)t_{private} + sh \times t_{shared}) \\
t_{private} &= h \times t_C + (1 - h)2t_{nw} \\
t_{shared} &= r \times t_{shared-read} + w \times t_{shared-write} \\
t_{shared-read} &= (h \times P_{\text{invalid}} + 1 - h)(t_{\text{read-miss}}) + h(1 - P_{\text{invalid}})t_C \\
t_{shared-write} &= (h \times P_{\text{not-dirty}} + 1 - h)(t_{\text{write-miss}}) + h(1 - P_{\text{not-dirty}})t_C
\end{align*}
\]

where \( t_X \) is a time needed for an \( X \)-type memory access. \( P_{\text{invalid}} \) is the probability that a shared variable in the cache of a given processor has been invalidated since it was last accessed by the same processor. This invalidation(s) results in changing the state to invalid, which would have been valid otherwise. Likewise, \( P_{\text{not-dirty}} \) is the probability that there has been an external access(s) to a shared variable after the last write to the variable by a given processor. This external access(es) results in changing the dirty state to valid or invalid, thus causing a write miss.

The time interval between two shared accesses from a given processor is \( t/sh \) and the probability that another processor issues a shared write during the time interval is \( 1 - e^{-\lambda t/sh} \) where \( \lambda \) is the mean arrival rate of shared writes from a given processor, i.e., \( sh \times w/t \). Since there are \( k \) shared variables which can be accessed by any processor, the probability that the write is to the same shared variable is \( 1/k \). Since the invalidation can be from any of \( n - 1 \) processors,

\[
P_{\text{invalid}} = 1 - (1 - \frac{1}{k}(1 - e^{-w}))^{n-1}
\]
Similarly, the time interval for two consecutive writes from a processor is \( t/(sh \cdot w) \) and the mean arrival rate of shared accesses is \( sh/t \). Therefore,

\[
P_{\text{not-dirty}} = 1 \cdot \left(1 - \frac{1}{k} (1 - e^{-1/w})^{n-1}\right)
\]

If false sharing is considered both \( P_{\text{invalid}} \) and \( P_{\text{not-dirty}} \) would be higher.

The time for servicing a read-miss, \( t_{\text{read-miss}} \), is \( 2t_{nw} + t_D \) if the main memory has a valid block, and \( 4t_{nw} + 2t_D + t_C \) otherwise. The main memory has a valid copy if the last global operation on that block was a read. So,

\[
t_{\text{read-miss}} = r(2t_{nw} + t_D) + w(4t_{nw} + 2t_D + t_C)
\]

On the other hand, \( t_{\text{write-miss}} \) is \( 4t_{nw} + 2t_D + t_C \) if the block is in dirty state in another cache, and \( 2t_{nw} + t_D + v(2t_{nw} + 2t_D + t_C) \) if there exist \( v \) valid copies. So,

\[
t_{\text{write-miss}} = r(2t_{nw} + t_D + v(2t_{nw} + 2t_D + t_C)) + w(4t_{nw} + 2t_D + t_C)
\]

Let \( m_X \) be a number of messages generated by a \( X \)-type memory access. Then we can get the following equations by the same way as for the timing analysis.

\[
total\ -\ messages = N((1 - sh)m_{\text{private}} + sh \cdot m_{\text{shared}})
\]

\[
m_{\text{private}} = 2(1 - h)
\]

\[
m_{\text{shared}} = r \cdot m_{\text{shared-read}} + w \cdot m_{\text{shared-write}}
\]

\[
m_{\text{shared-read}} = (hP_{\text{invalid}} + 1 - h)m_{\text{read-miss}}
\]

\[
m_{\text{shared-write}} = (hP_{\text{not-dirty}} + 1 - h)m_{\text{write-miss}}
\]

\[
m_{\text{read-miss}} = 2r + 4w
\]

\[
m_{\text{write-miss}} = r(2v + 2) + 4w
\]

In the following subsections, we apply the cost functions developed thus far to several widely used structures of parallel programs.
Parameters | value
---|---
h | 0.95
r | 0.70
\(t_{nw}\) | \(\log_2 N\)
\(t_{D, TC}\) | 1
\(t_{m}\) | 4 (=B)
v | 2

Table 3: Fixed parameters

5.4 Performance for Different Program Structures

In this subsection, we analyze the performance of cache schemes for programming paradigms which have been widely accepted in the parallel programming community. The first model represents a dynamic scheduling paradigm believed to be the kernel of several parallel programs [Pol88]. The basic granularity is a task. A large problem is divided into atomic tasks, and dependencies between tasks are checked. Tasks are inserted into a work queue of executable tasks honoring such dependencies. If a new task is generated as a result of the processing, it is inserted into the queue. All the processors execute the same code until the task queue is empty or a predefined finishing condition is met. A barrier is used to synchronize all the processors after executing a task. Correct queue operations require each queue access be atomic, and thus lock operations are needed for accessing the queue. Since all the processors are synchronized by the barrier, these lock requests are generated at the same time. The second model is the same as the first one except that the barrier synchronization is not used. The third model represents a static scheduling paradigm. In this model, the computation consists of several phases and each phase depends on the result of previous phase. Therefore, a barrier is used between phases. Table 3 shows values for fixed parameters.

Figure 5-8 show the effect of \(sh, k, N,\) and \(n\) on the completion time and the number of network messages for the WBI protocol and our scheme. In the legends, QB denotes the task queue with a barrier model, Q is the task queue without a barrier model, and S is the static scheduling case. Figure 5 shows that the completion times of both schemes are not affected much by \(sh,\) the degree of sharing. The WBI protocol is inferior to our scheme in all the tested cases. The completion time
for the WBI protocol increases as \( sh \) increases because it takes longer time for accessing shared variables than accessing private variables. But our scheme is not affected by \( sh \) since coherency maintenance is not necessary for shared variables. The huge performance gap between two cache schemes for the task queue with a barrier model comes from the overhead of parallel locks. The task queue alone (Q) or the barrier alone (S) does not hamper the performance that much.

Figure 6 shows the effect of different \( k \) values. As \( k \), the number of shared variables, decreases, the contention for share variables increase. The WBI protocol shows a slight decrease in the completion time and the number of messages generated as the value of \( k \) increase. For a fixed set of parameter values (\( sh = 0.3, k = 30, N = 300, \) and \( n = 32 \)), time for accessing shared variables is 1315 and the total time for memory accesses is 1619.62 while the total execution time is 3294 for Q model. That shows almost half the execution time is spent for synchronization activities. Therefore, even the memory access time is affected considerably by \( k \), the total execution time is not. The execution time of QB model is worse: more than 80% of the execution time is spent for synchronization.

The effect of the size of granularity in Figure 7 shows that the time is increased less than linearly and the number of messages increases slightly. That confirms the fact that the execution time is governed by synchronization overhead not by ordinary memory accesses. Otherwise the time should have increased linearly. The less increase in the number of messages can be explained by the fact that the ordinary memory accesses generate only a small portion of (usually, 3% for QB, 40% for Q and S) the total messages generated.

The question of scalability is answered in Figure 8. As the number of processors increases, the QB model shows sharp increases in both the completion time and the number of messages. That is because the QB model incurs \( O(n^2) \) overhead for both the metrics (see Table 1 and 2. For our scheme the QB model generates \( O(n^2) \) messages and \( O(n) \) takes completion time. That explains the steep increase in the number of messages for our scheme.

6 Concluding Remarks

For multiprocessors to be scalable, they should be able to tolerate large memory latencies as well as hop-spot contention. This paper shows that the most scalability issues can be handled by coherent
Figure 5: Effect of $sh$

Figure 6: Effect of $k$
Figure 7: Effect of $N$

Figure 8: Effect of $n$
private caches. The weak coherence model reduces global accesses and makes fast the execution of critical sections. And, thus the overall memory latencies are reduced. The memory latencies are enlarged if the load on the interconnection network becomes large. Reader-initiated coherence minimizes the transactions required to maintain the cache consistency. Thus, coupled with the cache-based lock scheme, the new cache scheme will enable the system to scale to a large number of processors. Most of the new cache primitives of our cache scheme will be used by the compiler or the programmer. Thus, the future research includes the program analysis for detecting the need for our primitives. Trace-driven simulation is also being performed to verify the analytical model used for performance evaluation.

References


Abstract

Clouds is a distributed operating systems research project at Georgia Tech. With threads and passive objects as the primary building blocks, Clouds provides a location-transparent protected procedure-call interface to system services. Mechanisms for synchronization within objects, and atomicity of computation are supported in Clouds. The primary kernel mechanism in object-based systems such as Clouds is the mapping of the object into the address space of the invoking thread. Hence the performance of such systems depends crucially on the efficiency of memory mapping. The problem gets exacerbated with distribution since now the object invoked by a thread may be located on a remote node. Since a thread can potentially invoke any object, the virtual address spaces of all objects can be viewed as constituting a "global distributed shared memory". Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-mapping mechanism. We present an organization and mechanisms for supporting this abstraction of a distributed shared memory. We propose a distributed shared memory controller that provides mechanisms for efficient access and consistency maintenance of the distributed shared memory. The novel feature of our approach is the exploitation of process synchronization to simplify consistency maintenance. The distributed shared memory mechanisms serve as the backbone for implementing object invocation, synchronization mechanisms, and network-wide memory management in the Clouds system.

1 Introduction

We are exploring hardware support to improve the performance of object-based distributed operating systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network. There are one or more processors and a certain amount of memory in each node. Nodes do not share memory; message exchange across the network is the only mechanism for communication between them.

Many operating systems designs for such an environment [4, 14, 20, 32] place a message-passing kernel on each node, supporting processes and communication between them via explicit messages. This kernel supports both local communication—communication between processes on the same node—and non-local or remote communication (sometimes implemented via a distinguished network manager process). Access to system services are requested via protected procedure calls in a traditional system, whereas in a message-based operating system they are requested via message passing. While a simple procedure call costs just a few instructions, and a protected procedure call (kernel call) costs a few hundred instructions, IPC costs a few thousand instructions in several systems that we studied [28].

Message-based operating systems are attractive for structuring distributed systems due to the separation of policy (encoded in server procedures) from mechanism (in the kernel).

Object-based distributed operating systems [7, 2, 39, 34, 26] view the resources of the system as a collection of objects. Clouds [7] is an object-based distributed operating system being developed at Georgia Tech. In Clouds, system services are encoded in passive objects (syntactic units that are similar in flavor to the server processes of message-based systems) that occupy distinct virtual address spaces in the system. Access to system services are requested by invocations (similar to the protected procedure calls of traditional systems) into the appropriate system object. The speed of object invocation is often used as a yardstick for measuring the performance of object-based systems. In passive object-based systems, invocation performance depends on the efficiency of object memory management (see §2). The problem gets exacerbated with distribution since now the invoked object may be located on a remote node.

In this paper we suggest mechanisms (that can be implemented in hardware/hardware) for supporting the abstraction of "globally distributed shared memory". In §2, we give the relevance of our work and motivate the need for supporting this abstraction. Related work in the area of memory coherence is presented in §3. With the Clouds operating system as our target application, we present our ideas on customizing the memory coherence requirements in §4. Our proposed hardware organization, the primitives provided by the distributed shared memory controller, and the algorithms for maintaining the consistency of the distributed shared memory are discussed §5. In §6 we describe a software implementation of the proposed primitives. A performance evaluation of our scheme is presented in §7. Finally, our conclusions are presented in §8.

2 Relevance

Clouds [7] is a distributed operating system that is intended to provide a unified environment over distributed hardware. Location independence for data as well as processing, atomicity of distributed computation, and fault-tolerance are some of the research goals of Clouds.

Objects and threads are the basic building blocks of Clouds. Objects are passive entities and specify a distinct and disjoint piece of the global virtual address space that spans the entire network. An object is the encapsulation of the code and data needed to implement the entry points in the object. Thus a Clouds object can be considered syntactically equivalent to an abstract data type in the programming language patience. Access to entry points in the object are accomplished through a capability mechanism in the kernel.

Threads are the only active entities in the system. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object. A thread enters an ob-
A thread enters an object by invoking an entry point in the object. It then executes the code in the entry point, and returns to the caller object. Binding the object invocations to the entry points in the object takes place at execution time. A thread in the course of its computation traverses the virtual address spaces of the objects that it invokes. A thread is comparable to a process (as defined in many conventional systems), with the exception that a thread may span machine boundaries. For the purposes of this paper, a process is synonymous to a thread.

The virtual address spaces of all objects can be viewed as constituting a "globally distributed shared memory". Such a view attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-mapping mechanism. Local object invocation involves mapping required memory segments of the object into the address space of the invoking thread. The current trend in structuring distributed systems is to use a collection of diskless computational servers (workstations) and a few data servers (file servers). Such an environment, the code and data for the (local) invocation has to be paged-in from the data server. Further, for one object invocation we have one of two choices: The first choice is to perform the computation at the node where the object resides (remote procedure call). The second choice is to have the invocation appear local by bringing in the segments needed for the invocation. While we have to support the for-immovable objects (such as an object that reads disk is), we believe that the latter may be a better choice for more objects. There are two reasons to support this belief:

the principle of locality [15] that suggests an invocation (or other invocations in the same object) may be repeated in the reduction in computational overhead due to the elimination of slave process management to support remote invocation at the node where the object resides [23].

This idea of using the abstraction of a globally distributed memory in an object-based system appears to be novel.

Other researchers have proposed software architectures on the shared memory paradigm, in different settings:

In a speech recognition application, Bisiani and Forin [9] use data structures that are shared by multiple language modules that are distributed on heterogeneous machines. They show that communication through shared memory is viable alternative to message-passing even when the environment involves cooperation between multilingual program modules and heterogeneous machines.

Processes in the programming language Linda [10,18] communicate via a globally-shared collection of ordered tuples. Logically shared bulletin-board is proposed by Birman, et. al. [8] for structuring asynchronous interactions between processes in distributed systems.

Exploit the multicast capability of local area networks, Mad and Bernstein [1] describe a bulletin-board based process communication mechanism that could just as well be implemented with a shared memory paradigm.

Integrating the mechanisms for virtual memory management and local interprocess communication, Mach [32] achieves efficient implementation of local interprocess communication. Currently, researchers at CMU are investigating the duality of shared memory and message passing in the context of network communication as well [36].

[39] achieves substantial reduction in the cost of programming by using copy-on-write techniques (Accent and on-demand fetches during remote execution.

- Emerald [21] is a distributed object-based language and system with support for object mobility.

We believe that software architectures based on the shared memory paradigm would benefit considerably (in both performance and ease of implementation) if the underlying hardware were to provide a transparent mechanism for efficient access and consistency maintenance of distributed shared memory. Thus the investigation of hardware support for providing the abstraction of a distributed shared memory is worthwhile.

3 Related Work

3.1 Shared Memory Multiprocessors

Consistency maintenance in distributed shared memory is similar to cache coherence in multiprocessors. Shared memory multiprocessors such as Encore's Multimax [16], consist of several processors connected to a common shared memory via a system bus. A main memory cache is associated with each processor to help reduce the traffic to the shared memory. Multiprocessor cache consistency protocols (such as [19,22]) ensure the following memory coherence constraint: a read operation performed by a processor returns the most recent value written into that location (by any processor). This criterion is appropriate in a shared memory multiprocessor since the system bus (a broadcast medium) serialized the memory operations of all the processors. This approach is a brute-force one to assure coherence since there is no semantic knowledge associated with the data being cached.

We make the following observations regarding these protocols: Multiprocessor cache coherence algorithms (see [3] for a survey) consider memory coherence problem in isolation. In reality, memory coherence and process synchronization are closely intertwined. The ability of a process to read or write shared data is invariably acquired through some synchronization method. Since the cached data has no synchronization information associated with it, these algorithms tend to be an overkill owing to their generality. However, these algorithms are a viable approach for solving the cache coherence problem in multiprocessors since the cost (measured in circuit complexity as well as time) of implementing them in hardware is a small fraction of the total system cost. Further, multiprocessors have the ability to invalidate all cached copies in one atomic bus cycle owing to the system bus.

3.2 Distributed Shared Memory

Some work has been done in extending the shared memory paradigm to a distributed system. Li [24] presents a slight variation of the Berkeley protocol for multiprocessor cache consistency [22] as a solution to maintain the consistency of distributed shared memory. The entire memory is considered potentially sharable for both reads and writes. Hence the current owner of a page (the node that has write access to the page) keeps a copy-set—the set of nodes that have a read-copy of the page. In Li's scheme, a write into a shared location results in an invalidation message to be sent to each of the nodes in the copy-set for the page containing the location. Note that the nodes that have a copy may no longer be interested in that page. These invalidation messages are a high price to pay in a distributed environment. Some of the pages may never be written into (for example text pages), but since the algorithm only deals with raw pages the overhead of
keeping the copy-set information is also incurred for such pages. Li's solution has the same drawback as the multiprocessor cache coherence algorithms—memory coherence problem is dealt with in isolation without considering process synchronization.

Agora [9] (see §2) supports shared data structures that span heterogeneous machine architectures and multiple languages. Agora adopts a mechanism similar to Li's with invalidation messages on writes to shared locations, and hence is inefficient for the same reason. However, since Agora is tailored for a very specific application it allows sharing at the level of individual data structures rather than raw pages.

Fleisch [17] proposes a distributed shared memory facility for the Locus [26] system that supports Unix System V shared memory semantics. The proposed coherence algorithm is similar to Li's [24]. Fleisch's work does not address such issues as locality and synchronization.

4 Customizing Coherence

We are interested in exploring hardware support for the abstraction of a distributed shared memory. The basic idea is the following: Each node has associated with it a "network cache"—a repository for recently accessed remote memory-segments and their owners (nodes). If a remote memory-segment is not in the cache, it is requested from the owner and cached for future reference. If the segment is "dirtied", the network-cache becomes the supplier in the future for this segment. If and when the segment is replaced it is sent back to the owner of the segment.

How do we define memory coherence in this environment? The definition that works well for a shared memory multiprocessor is inappropriate for this environment since there is no "system bus" to impose a total order on the memory operations that are performed by all the processors. Further, while invalidation of cached copies of data is a viable approach in multiprocessors (with a system bus) it is infeasible (due to the cost of the invalidation messages) in a distributed system. Invalidation involves at least sending a multicast message to all the nodes that have a read-copy of the segment. Achieving reliable delivery of such multicast messages is prohibitively expensive in a distributed system [13].

Exploiting application specific semantic information would reduce the complexity of the problem and make a hardware solution viable. In the scheme that we propose we deal with process synchronization and memory coherence together. Since the application that we are trying to support is the Clouds operating system we use the structure of the objects to dictate the coherence requirement. However, since Clouds is a general purpose operating system we cannot make any more assumptions as Agora does in providing finer-grain sharing at the level of individual data structures. Read-only code and data areas can be distributed without the need for maintaining consistency. Only the read-write data area of an object requires maintenance of consistency.

5 Hardware Organization

The organization (Figure 1) we propose inside each node of the network is the following: a host that executes distributed applications; a distributed shared memory controller (DSMC) together with the network interface assists the host in mapping memory segments (local and remote) into the virtual address spaces of the application processes. There is a minimal kernel on the host that traps system calls, and virtual address translation faults. The DSMC is also in control of the network. The system memory is (logically) partitioned into two parts: One part (object memory) is for housing the segments of locally created objects; the other part (network cache) is for caching segments from remote objects. Conceptually there are two lists of process control blocks: the host-list that the host looks at to schedule runnable processes; the DSMC-list that the DSMC looks at to service memory segment requests (local and remote). The host-enqueues processes that fault on virtual address translation in the DSMC-list. The DSMC enqueues processes that have become runnable again (after the fault service) in the host-list. The operations provided by the DSMC, and the algorithms for implementing these operations are the topics of interest in this paper. Although conceptually the DSMC is shown as a "co-processor", it can be implemented as a software module that coexists with the kernel (see §6).

5.1 Clouds Objects

Objects in Clouds consist of one or more of the following types of areas: read-only code, read-only data, shared read-write data. We refer to these areas as segments. Segments serve as containers of data and can be of variable size. The contents of a segment may only be accessed when the segment is attached to an object [6]. Segments persist until explicitly destroyed. We refer to the node where a segment is created as the owner of the segment. There is a well-known area of the system memory—keeper segment—maintained by the DSMC. We associate a keeper location with each read-write segment owned by the node. The keeper location for a segment points to the node currently having write access to the segment. Owner and keeper point to the same node at segment creation time.

5.2 Virtual Address

The virtual address generated by a process is interpreted as being composed of three fields: object name, segment name, and segment offset. Using the segment name, the DSMC does a table lookup to determine the location of the segment\footnote{The location table is maintained by the kernel. If the segment name does not appear in the table, the DSMC can request the kernel to locate the segment.}. The DSMC at each node keeps a segment table. This table contains the following information for segments currently mapped at the node: their sizes and types, and their mapping to physical segments or disk blocks. Further, for locally created segments the segment...

Figure 1: Hardware Implementation
be remembered the current location (keeper) of the segment. A segment-table entries also each have a queue—the list of processes waiting on the segment.

A segment fault occurs when the virtual address generated by a process is currently unmapped. In addition, every object location results in a fault to map in the segments needed for invocation. A segment fault (or object invocation) manifests when a process calls an object manager that in turn invokes the DSMC routines. The host enqueues the faulting process in the DSMC-list (for the DSMC), and schedules the next runnable process. DSMC is responsible for mapping in the required segments making the process runnable. A process waiting for a remote segment is queued on the appropriate entry in the map table. Receiving a remote segment in response to a previous request, DSMC sets up the memory map of the processes waiting for the segment and enqueues these processes in the host-list (for use).

Although the segment as used in this paper is a logical entity, it is paged onto the underlying hardware into an integral number of pages. In Reference [6], we discuss an implementation of the DSMC as a software module that is part of the paging system. Implementation is summarized in §6.

**DSMC Primitives**

DSMC provides data transfer and synchronization primitives supporting the abstraction of a global distributed shared memory. From the applications (Clouds kernel in our case) entertains four types of requests:

- **Get(segment)**: The DSMC is responsible for fetching the requested segment, setting up the segment table of the faulting process, and enqueuing the process on the host-list for execution.

- **Lock(segment)**: The DSMC frees the physical memory occupied by the segment by sending the segment back to the owner.

- **Send(segment, semaphore)**: The DSMC performs an atomic semaphore P operation on the specified synchronization variable.

- **Send(segment, semaphore)**: The DSMC performs an atomic semaphore V operation on the specified synchronization variable.

ICs exchange messages to satisfy these requests. The recognized by the DSMC are the following:

- **Get(segment)**
- **Send(segment)**
- **Lock(segment)**
- **Send(segment, semaphore index i)**
- **Send(segment, semaphore index i)**
- **Send(segment, semaphore index i)**
- **Send(segment, error type)**

In the next few subsections, we give the algorithms required for the data transfer and synchronization primitives of understanding the algorithms, we show a single execution while processing these primitives. However, it is noted that the DSMC is multiprogrammed. For every request the local DSMC sends a msg_get request to its peer and then proceeds to the next request in its queue (DSMC-list). Eventually, when a reply arrives from its peer the local DSMC takes the appropriate action. In the algorithms to follow, we do not show this asynchrony. For simplicity, we show each request from a DSMC to its peer as a synchronous one.

### 5.3.1 Data Transfer

**Get(segment)** is the primitive for mapping a currently unmapped segment. On a segment fault the host enqueues the faulting process in the DSMC-list with a get request for the required segment. A get request for a read-only segment is trivially satisfied if the object is remote; if the object is local; if the object is remote the DSMC requests its peer (at the owner node) for the read-only segment. On receiving the request, the peer DSMC sends the read-only segment to the requesting DSMC without performing any housekeeping work.

A get request for a read-write segment is implemented by the following algorithm:

```
if local(segment) then
    case keeper of
        self:
            if not memory(segment) then
                bring segment into object-memory from disk;
            end;
        remote:
            send msg_get(segment) to remote DSMC;
            receive msg_segment(segment) from remote DSMC;
            place segment in object-memory;
            change keeper to self;
            endcase;
    else /* segment is remote */
        /* go ask the owner */
        send msg_get(segment) to remote DSMC;
        receive msg_segment(segment) from remote DSMC;
        place segment in network-cache;
    endif;
else /* the requested segment is now in memory */
    map it into the process' address space */
    map segment into process' address space;
```

When a DSMC receives msg_get request for a (read-write) segment it does the following:

```
if not memory(segment) then
    self:
    bring segment into object-memory from disk;
end;
change keeper to requesting DSMC;
send msg_segment(segment) to requesting DSMC;
endif;
remote:
    send msg_forward(segment) to keeper;
/* note at most one forwarding */
change keeper to requesting DSMC;
endcase;
```

The owner may forward the request to the current keeper. A DSMC that receives a msg_forward request for a (read-write) segment does the following:

```
send msg_segment(segment) to requesting DSMC;
invalidate the segment entry in the segment-table;
return freed segment in the network-cache to free-list;
```
Placing a segment in the network-cache may involve freeing up segments from the network-cache. The DSMC sends msg_discard to the owner DSMC of the segment for this purpose. The algorithm for freeing up a segment from the network-cache is the following:

- Invalidate the segment entry in the segment-table.
- If read-write(segment) then
  - If dirty(segment) then
    - Send msg_discard(segment) enclosing segment to owner.
  - Else /* clean segment */
    - Send msg_discard(segment) to owner.
- Endif.

The owner DSMC upon receiving msg_discard request does the following:

- If segment enclosed then
  - If memory(segment) then
    - Write segment into memory.
  - Else
    - Write segment onto disk.
  - Endif.
- Endif.
- Change keeper to self.

Due to network delays it is possible that a node (the future keeper of a segment) may receive a msg_forward request before the segment arrives from the owner or the forwarder (the current keeper). However, recall that the keeper information with the owner of an object is absolute. Therefore the node can simply buffer the segment-request, and honor it when the segment arrives, without compromising the correct operation of the system.

### 5.3.2 Synchronization

Get, and discard are not enough to efficiently implement the synchronization primitives provided by the Clouds kernel. For example, consider semaphore operations \((P, V)\) supported by the Clouds kernel for synchronization inside an object. With just get and discard operations supported by the DSMC, an obvious implementation of the semaphore operations may be the following: place the semaphore data structure (a value field, and a list of waiting processes) in a read-write segment; on every semaphore operation get this data structure from its keeper, and perform the operation atomically (disabling interrupts). By definition, semaphore operations order the execution of cooperating processes. Therefore, it seems wasteful to ship the semaphore data structure back and forth between these processes when they are on different nodes.

For now we consider the solution where the semaphore operations are performed at the owner node. The semaphore operations are provided as primitives understood by the DSMC. Synchronization variables are allocated semaphore segments. We note the fundamental difference between the roles played by the DSMC in the data transfer operations \((get, discard)\) and the synchronization operations \((P, V)\): In the former, the DSMC fetches the current copy from the keeper (it does not have to worry about the contents of the fetched segment); whereas in the latter the DSMC manipulates the contents of the synchronization variables at the owner node and reports success/failure to its requesting peer DSMC.

When a process performs an operation \((P \text{ or } V)\) on a synchronization variable, it results in a trap to the kernel. The kernel in turn enqueues this process in the DSMC-list. The algorithm executed by the DSMC for implementing \(P\text{(segment, semaphore i)}\) or \(V\text{(segment, semaphore i)}\) operation is the following:

- If local(segment) then
  - Case operation of \(P\):
    - Decrement synchronization variable \(i\).
    - If variable value less than zero then
      - Enqueue process on queue(segment, i).
    - Else
      - Enqueue process on host-list; /* ready to resume execution */
  - Endcase;
  - V:
    - Increment synchronization variable \(i\).
    - If variable value less than or equal to zero then
      - Remove an entry from queue(segment, i).
  - Case operation of \(V\):
    - Send msg_ack(segment, i) to remote DSMC;
    - Enqueue process on host-list; /* ready to resume execution */
    - Endcase;
- Else /* remote segment */
  - Case operation of \(P\):
    - Send msg_P(segment, i) to remote DSMC;
    - Enqueue process on queue(segment, i);
    - Receive msg_ack(segment, i) from remote DSMC;
    - Dequeue process from queue(segment, i);
    - Enqueue process on host-list; /* ready to resume execution */
  - Endcase;
  - V:
    - Send msg_V(segment, i) to remote DSMC;
    - Enqueue process on host-list; /* V-ing process */
  - Endcase;

On receiving a msg_P or a msg_V request the DSMC does the following:

- Case operation of \(P\):
  - Decrement synchronization variable \(i\).
  - If variable value less than zero then
    - Enqueue (requesting DSMC, process) on queue(segment, i).
  - Else
    - Send msg_ack(segment, i) to requesting DSMC;
  - Endif;
  - V:
    - Increment synchronization variable \(i\).
    - If variable value less than or equal to zero then
      - Remove an entry from queue(segment, i).
  - Case entry of remote DSMC:
    - Send msg_ack(segment, i) to remote DSMC;
    - Enqueue process on host-list; /* removed entry */
  - Endcase;
- Endif;

5.3.3 Merging Data Transfer and Locking

Our data transfer primitives eliminate invalidation messages by keeping exactly one copy of a read-write segment. However.
 Keeping just one copy reduces availability (for readers) in applications that can be modeled as a readers/writers problem. The fact that there is exactly one copy of a read-write segment is appropriate from the point of view of the writers while being a severe restriction for the readers. Since we expect such applications to be encountered quite frequently, we propose the following modification to the get primitive to increase the availability. The modification is to include mode information in the primitive: \texttt{get(object, segment, mode)}, where mode can be one of read-only, read-write, or none. The DSMC keeps two additional pieces of information in the segment table for read-write segments: lock mode, and readers. When the owner DSMC receives a request for a read-write segment it does the following:

```plaintext
if queue(segment) empty then
  case keeper of
    self:
      if not memory(segment) then
        bring segment into object-memory from disk;
      endif;
      change keeper to requesting DSMC;
      send msg_segment(segment) to requesting DSMC;
      remote:
        send msg_forward(segment) to keeper;
      /* note at most one forwarding */
      change keeper to requesting DSMC;
    endif;
  endcase;
  return;
else /* mode <> none */ and (mode = none) then
  enqueue requesting DSMC on queue(segment);
  return;
endif;
if keeper is remote then
  * the owner remains the keeper when get requests are issued
  else /* first-entry.mode = read-only */
    if (lock-mode = read-only) then
      decrement readers;
      if readers = 0 then
        lock-mode = none;
      endif;
    elseif (first-entry.mode = none) and 
      (queue(segment) not empty) then
      send msg_error(segment, mode_conflict) to requesting DSMC;
      goto process_queue;
    endif;
    if not memory(segment) then
      bring segment into object-memory from disk;
    endif;
    change keeper to requesting DSMC;
    send msg_segment(segment) to requesting DSMC;
    remote:
      send msg_get(segment, none) to keeper;
    change keeper to self;
    if (lock-mode <> read-write) and (mode = read-only) then
      * readers can enter if lock-mode is either read-only or none */
      set lock-mode to read-only;
      increment readers;
    endif;
    send msg_segment(segment) to requesting DSMC;
    if (lock-mode = none) and (mode = read-write) then
      * a writer can enter if lock-mode is none */
      set lock-mode to read-write;
    endif;
    send msg_segment(segment) to requesting DSMC;
    queue requesting DSMC on queue(segment);
    if /* if (lock-mode <> read-write) */ 
    endif;
    if (lock-mode = none) then 
    if queue is non-empty */
      the kernel knows when to discard a segment? In readers/writers problem the reader (or the writer) explicitly acquires the appropriate lock, reads (or writes), and releases the lock. In Clouds, lock and unlock are system operations. When the Clouds application programmer uses these primitives, the operating system translates these primitives to get and discard the appropriate segments, respectively. We note that there are possibilities of deadlock if the application fails to release a lock. If an application fails to perform an unlock operation, the algorithm is in error. We envision a deadlock detection mechanism in the operating system that is layered on top of the DSMC mechanisms. The deadlock detector would issue unlock requests on behalf of the deadlocked processes. Note that if the kernel fails to perform a discard operation upon an unlock request, it is tantamount to a kernel bug.

With this enhancement our DSMC provides as much as availability as Li's algorithms [24] without incurring the high cost of invalidation messages. Note that explicit discards from the readers are in lieu of the invalidation messages. However, we contend that these are exactly the minimum number of messages required to maintain a consistent shared data structure.

5.3.4 Weaker Semantics

The above DSMC primitives provide strong memory coherence, where a write to a location by a process is seen by other processes when they access the same location. While this criterion is appropriate for applications that rely on the DSMC to enforce memory coherence, other applications may find these primitives too restrictive. Some applications may need to provide for memory coherence as part of their algorithms, while other applica-
tions may not need strong memory coherence. For example, a
system monitoring facility may need to inspect the contents
of some segments without acquiring locks, while a distributed
game that maintains the state of a graphics screen in shared memory
may sacrifice strong memory coherence for better performance.

For such applications, we provide a simple mechanism to ac-
quire a copy of a segment without enforcing memory coherence.
We define weak-read mode for the get request. A get(segment,
weak-read) request for a segment acquires a copy of the segment
from the owner DSMC. Upon receiving a get(segment, weak-
read) request, the owner DSMC sends a copy of the segment
to the requester, regardless of the fact whether a copy of the
segment exists in any other node.

5.3.5 Summary of Modes

In summary, using the get primitive a segment may be acquired
in one of four modes: read-only, read-write, weak-read, or
none. Read-only mode signifies non-exclusive access but guaran-
tees that the segment will not change until the node explicitly
discards the segment. Read-write mode signifies exclusive ac-
cess (for the node) with a guarantee that the segment will not
be thrown away until the node explicitly discards the segment.
Weak-read mode signifies non-exclusive access with no guaran-
tee whether the segment will change or not. None mode signifies
exclusive access with no guarantee whether the segment will be
taken away or not.

6 Implementation of DSMC

Ra [5,7] is an operating system kernel designed to be the nucleus
of Clouds operating system. It is currently implemented on the
Sun-3 architecture. Ra defines and manages three primitive ab-
stractions: segment, virtual space, and aruba. The contents of a
segment may only be accessed when that segment is mapped to a
range of addresses in a virtual space. Virtual spaces abstract the
notion of an addressing domain, and they are composed of seg-
ments. Ra isbas are an abstraction of the fundamental notion of
computation or activity and can be thought of as lightweight
processes.

The Ra kernel is responsible for mapping segments into vir-
tual memory using the memory management hardware provided
by the underlying architecture. The size of a segment is a mul-
tiple of the physical page size. Ra assumes the existence of
partitions that are responsible for realizing, maintaining, and
storing segments. Partitions are an example of system objects.
System objects encapsulate necessary and/or useful operating
system services and resource managers that have direct access
to the Ra kernel, but are nonetheless outside the kernel. System
objects are trusted software modules that are loaded dynami-
cally in the system space. Other system objects include device
drivers, resource managers, and user-level object support. Each
partition provides (at least) the following calls for use by Ra:
activate/deactivate segment, create/destroy segment, and page-
in/page-out portions of segments. When Ra is instructed to
serve a segment request (e.g. to map a segment into a virtual
space), it invokes the appropriate partition to fetch the segment
into physical memory. Ra then manipulates the memory man-
agement hardware to map the physical pages appropriately.

We have implemented the DSMC as a software module that
consists of approximately 3500 lines of C++ [34]. Figure 2
shows the organization of the DSMC implementation on Ra.
The boxes in the figure denote system objects. The DSMC
cooperates with remote DSMC's to implement the distributed
shared memory primitives. DSM Partition is a Ra partition

![Figure 2: Organization of DSMC implementation under Ra](image)

![Figure 3: Organization of DSMC implementation on Unix](image)
among a set of clients and servers. The DSMC code is linked-in with client code. DSMC code is also linked-in with server code that uses the Unix file system to store segments. Implementation of DSMC on Unix and Ra serves three purposes:

1. The Unix environment makes it easy to test and verify the DSMC and TAL protocols.
2. The Unix file system is available for use as permanent store for segments. Ra executes on diskless Sun-3 workstations with backing store provided by Unix machines.
3. The strength of Unix is the rich program development environment that it provides. The strength of Clouds is transparent management of distributed data and computation. Providing interoperability between Unix and Clouds is one of our design goals. DSMC implementation on Unix and Ra serves this purpose. System and user objects are developed on Unix and demand-paged to Ra via DSMC mechanisms.

The Unix implementation of the DSMC and TAL is complete, and we report on its performance in §7. The Ra implementation of the DSMC is awaiting the completion of an ethernet for Ra.

Performance Results

measured the performance of the DSMC implementation in Unix. All measurements are done on Sun-3/60 workstations 4M bytes of memory, connected through a 10M bits/sec net. We mask out the cost of secondary storage access by loading segments in memory before measuring the costs of the C primitives. Table 1 summarizes the results.

<table>
<thead>
<tr>
<th></th>
<th>without forwarding</th>
<th>throughput:</th>
<th>with forwarding</th>
<th>throughput:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get or discard (8K bytes)</td>
<td>43.4 ms</td>
<td>185 Kbytes/s</td>
<td>63.7 ms</td>
<td>126 Kbytes/s</td>
</tr>
<tr>
<td>V operation:</td>
<td>16.5 ms</td>
<td>23.3 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P operation:</td>
<td>31.5 ms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activate segment:</td>
<td>43.4 ms</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Measurements of DSMC operations on Unix

Table shows that on an average fetching a segment forwarding) of size 8K bytes (the page size on the Sun-434 ms. Van Renesse et al. report a transfer rate of 40 Kbytes between two user processes on different nodes in RPC on a 10M bits/sec ethernet [36]. Our implementation uses two user processes per node and still compares favorably with the figures reported by van Renesse et al. A null from one DSMC to another costs roughly 20 ms, a large fraction of which is spent context switching between the kernel, and between TAL and DSMC. Moving TAL into the kernel would eliminate the additional context switch.

we are currently investigating such an implementation. Also, the issuing process continues without waiting for acknowledgment from the remote DSMC.

we also evaluated our DSMC scheme and compared it using simulation [28]. In the simulation, the DSMC is a software module as described in §6. The performance shows that:

- Over a range of object invocation locality, DSMC has significant advantages over RPC.
- Use of distributed shared memory achieves automatic distribution of processing load, by performing the invocation locally instead of on the owner nodes.
- Instead of addressing memory coherence and synchronization separately, applications that can be modeled as readers/writers problems benefit considerably when locking and segment access primitives are combined.

As mentioned before, the DSMC primitives are provided to the operating system as a set of mechanisms for managing memory in the network. Policy decisions, such as when to use RPC and when to use the DSMC primitives, are in the operating system. The operating system decides how and when to use the DSMC primitives. For example, during periods of high concurrent access to an object, the operating system may use the DSMC primitives to locate the object at one node, and then use RPCs to invoke the object thereby reducing data movement across the network.

Comparison with Li’s Scheme

To illustrate the advantage of combining mutual exclusion and consistency maintenance, we show how a readers/writers problem can be implemented using our scheme and Li’s scheme, and compare the number of messages generated in each case. Li’s basic scheme can be summarized as follows:

- On a read fault:
  1. Ask manager for page.
  2. Manager forwards request to owner.
  3. Owner sends copy of page to requester.

- On a write fault:
  1. Ask manager for page.
  2. Manager forwards request to owner.
  3. Owner sends page and copy-set to requester.
  4. Requester invalidates all copies in copy-set by sending a message to each node holding a read-copy of the page.

Suppose we want to program the following readers/writers problem: A segment that is accessed by a set of readers and writers resides on the manager node M, and each of the readers/writers run on a different node (for simplicity, we assume that no reader/writer runs on the manager node). Each reader/writer computes for a while, then accesses the shared segment. It is clear that some mechanism to synchronize access to the segment is needed.

In our scheme, locks (see §5.3.3) can be used to solve this problem as follows:

Reader:

```c
loop
  compute for a while
  lock segment in read-only mode
  /* lock generates one message to manager */
  /* manager eventually sends back segment */
  access segment
  unlock segment
  /* unlock generates one message to manager */
endloop
```

```c
```
Similarly for a writer:

Writer:

loop
    compute for a while
    lock segment in read-write mode
    /* lock generates one message to manager */
    /* manager eventually sends back segment */
    access segment
    unlock segment sending modified segment back to manager
    /* unlock generates one message to manager */
endloop

It is not clear how one can program this simple readers/writers problem using Li's primitives, for Li does not address the issue of process synchronization. Process synchronization has to be addressed separately, because there is no way within Li's scheme for a user to lock a page while accessing it. Therefore, we assume that a lock operation is implemented by sending a message to a distinguished server, requesting access to the shared segment in the required mode. When the reply is received from the server, the shared segment is accessed. When the reader/writer is finished accessing the segment, it sends an unlock message to the server.

Using our scheme, the lock operation generates 2 messages: one message to the manager, and another from the manager to the requester (the second message includes the segment). The unlock operation generates one message to the manager (a discard message that includes the segment in the case of an unlock by a writer). Accessing the segment generates no messages, because the segment is made available locally as a result of the lock operation. Using Li's scheme, the lock operation generates 2 messages: one message to the server, and another from the server back to the requester (indicating that the requester can now access the segment). The unlock operation generates one message to the server. However, accessing the segment may generate several messages. On a read-fault, 2 messages are generated: one to the current holder of the segment, and another from the holder to the requester (the second message includes the segment). In addition to the 2 messages required to bring the segment from the current holder to the requester, a write-fault also generates \( O(r) \) invalidation messages, where \( r \) is the number of readers. The invalidation messages are required because process synchronization is separated from consistency maintenance—the fact that the lock server gave permission for access to the segment is unknown to the consistency maintenance algorithm. The table below summarizes the comparison in terms of the number of messages:

<table>
<thead>
<tr>
<th>Number of messages</th>
<th>Reader</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our Scheme</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Li's Scheme</td>
<td>5</td>
<td>( 5+O(r) )</td>
</tr>
</tbody>
</table>

8 Conclusion

The abstraction of distributed shared memory is attractive from the point of view of object-based systems such as Clouds as well as other software architectures that use the shared memory paradigm for process communication. Such architectures would benefit considerably if the underlying hardware were to provide some transparent mechanism for efficient access and consistency maintenance of the distributed shared memory. We presented an organization and mechanisms for supporting this abstraction. The novel features of our approach are the use of distributed shared memory as an alternative to RPC, and the exploitation of process synchronization to simplify consistency maintenance.

References


II-169
A Measurement-based Study of Hardware Support for Object Invocation

UMAKISHORE RAMACHANDRAN AND M. YOUSEF AMIN KHALIDI
School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332-0280, U.S.A.

SUMMARY

The object invocation paradigm is attractive for structuring distributed systems. Distributed object-based operating systems view the resources of the system as a collection of objects. Object invocation is the primary mechanism in such systems, and is often used as a yardstick for measuring the system performance. However, existing systems of this flavour exhibit poor performance due to the mismatch between the requirements of the object invocation mechanism and the machine architecture. Through measurements of an existing object-based kernel, we present a breakdown of the costs involved in implementing the object invocation mechanism. The measurements suggest architectural solutions to improve the performance of such systems. We present our preliminary studies towards providing hardware support for the object invocation mechanism.

KEY WORDS Object invocation Measurements Hardware support Distributed operating systems

INTRODUCTION

This paper explores architectural support to improve the performance of object-based distributed operating systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network. There are one or more processors and a certain amount of memory in each node. Nodes do not share memory; message exchange across the network is the only mechanism for communication between them.

Operating system structures for a distributed environment follow one of two paradigms: message passing or object invocation. Message-based operating systems place a message-passing kernel on each node, supporting processes and communication between them via explicit messages. This kernel supports both local communication—communication between processes on the same node—and non-local or remote communication, sometimes implemented via a distinguished network manager process. In a traditional system such as Unix, access to system services is requested via protected procedure calls, whereas in a message-based operating system it is requested via message passing. Message-based operating systems are attractive for structuring distributed systems due to the separation of policy, encoded in server processes, from mechanism in the kernel.
Object-based distributed operating systems view the resources of the system as a collection of objects. Objects are similar to abstract data types, and are written as individual modules composed of the specific operations that define their interface. Access to system services is requested by invocation on the appropriate system object. In this sense, object-based distributed operating systems combine the advantages of both the traditional systems and the message-based systems. The invocation mechanism is similar to a protected procedure call, and objects encapsulate functionality similar to the server processes of message-based systems. Object invocation is the fundamental facility in object-based systems, and the speed of object invocation is often used as a yardstick for measuring the performance of such systems. The objective of this research is to understand the costs incurred in object invocation, and to propose hardware and software mechanisms to reduce these costs, and hence improve the performance of object-based systems.

Clouds is used as the test bed for evaluating our research ideas. Although the issues investigated in this research are presented in the context of Clouds, they are general and apply to other object-based systems.

Clouds is an object-based distributed operating system being developed at Georgia Tech. Objects and threads are the basic building blocks in Clouds. Objects are passive and persistent entities in the system. An object is the encapsulation of the code and data structures needed to implement the entry points in the object. These entry points provide the procedural interface for an activity to execute the code in an object. This code may itself call entry points in the same or other objects. Objects are disjoint partitions of a global virtual space that spans the entire network. Objects consist of one or more of the following types of areas: read-only code, read-only data, shared read-write data. These areas are referred to as segments.

Threads are the only active entities in the system. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object. A thread enters an object by invoking an entry point in the object. It then executes the code in the entry point, and returns to the caller object. The binding of an object invocation to an entry point in an object takes place at execution time, and more than one invocation may execute in the same object concurrently. Figure 1 shows the model of computation in Clouds. A thread is comparable to a process as defined in many conventional systems, with the exception that a thread may span machine boundaries. For the purposes of this paper, a process is synonymous with a thread.

![Figure 1. Model of computation](image-url)
A performance study of several object-based operating systems is desirable to ensure that we are not discovering coding inefficiencies of one operating system, but instead see a trend that is common to all systems. Unfortunately, there are very few systems, implemented on bare hardware, that can be helpful in such a study. Alpha and Clouds are appropriate systems to study, but we only have access to the Clouds kernel. Therefore, the Clouds kernel is used as the system to study, but necessary care must be taken so as not to draw the wrong conclusions when analyzing the Clouds implementation. The implementation may hide the real costs of object invocation. Therefore, a two-step strategy is followed:

1. Extensive profiling of the current implementation is done. Implementation inefficiencies are identified, and modifications to the kernel are made to remove these inefficiencies.
2. The modified kernel is then analyzed, and costs intrinsic to object-based systems implemented on conventional machines are identified.

To identify and eliminate all inefficiencies amounts to rewriting the kernel. Therefore, an effort is made to remove glaring inefficiencies and to conduct the experiments in such a manner as to exercise only the portions of the kernel that we are interested in. For example, by bringing objects into main memory before starting an experiment, the performance of secondary storage is masked out. It is to be noted that a new implementation is currently under way based on our experience with the existing system. The new implementation is discussed in a subsequent section.

Overview of the invocation mechanism in Clouds

Local invocation in passive-object systems involves mapping the required memory segments of the object into the address space of the invoking thread. Remote invocation is implemented as a local invocation at the remote node where the invoked object resides, similar to a remote procedure call.

The Clouds kernel is implemented on the Vax-11 architecture. The Vax architecture provides paged virtual memory, with the system kernel mapped into one half of the memory space (system space) and the currently executing process mapped into the other half (process space). The process space is further divided into the Vax-designated P0 and P1 spaces. Program images and most of their data reside in P0 space, whereas P1 space contains the process stacks and other data. A page table that resides in physical memory describes the system space, and two separate page tables describe the currently mapped process space. The process page tables reside in system space virtual memory. The hardware provides a translation look-aside buffer (TLB) for caching recent virtual-to-physical memory translations. The TLB is split into two halves: one half caches translations from the system space, whereas the other half caches translations from the process space. The latter part is invalidated on each process context switch.

Local object invocation is considered first. When a thread t executing in object O1 invokes an entry point in object O2, the following steps are taken: the calling object O1 constructs two argument lists (arglists), one for transferring arguments out, and the other to receive output parameters from object O2. Each arglist consists of a count field, an overflow indicator field, and an array of argument descriptor records. Each
data descriptor record specifies one argument or a pointer to an argument (see Reference 12 for more details). After constructing the arglists, thread \( t \) enters the kernel through a protected system call (trap). The parameters to the system call include the starting addresses of two arglists, a capability to object \( O_2 \), and the number of the entry point to invoke in \( O_2 \). The kernel then calls the \textsf{ObjInvoke} routine that uses the capability to search an in-memory system table for \( O_2 \). If a descriptor for \( O_2 \) is not found, local secondary storage is searched. When \( O_2 \) is found, its descriptor is read into memory, and the information contained in the object's descriptor is used to construct a P0 page table for the object. The kernel then copies the arguments onto the process stack, allocates an \textsf{ObjRec} structure that holds information about \( O_1 \), and links it to the process control block (PCB) in a LIFO manner to be used later during object return. A new mapping of the P0 space is set up, and a process context switch that actually maps \( O_2 \) into P0 is executed. A side-effect of the context switch is the invalidation of the process's TLB. Execution continues in the \textsf{ObjInvoke} routine which returns to the trap handling routine, which in turn 'returns' to object \( O_2 \) to start the invocation.

Before \( O_2 \) returns to \( O_1 \), an arglist of return parameters is constructed. The thread \( t \) then enters the kernel through the protected system call interface, passing back the starting address of the return arglist and a success/error indication. The kernel calls \textsf{ObjReturn} to perform the following steps: the first \textsf{ObjRec} is dequeued from the PCB, and the information contained in it is used to locate \( O_1 \). The return parameters are copied into a temporary area, a new mapping of the P0 space is set up, and a context switch is executed. The return parameters are then copied into the locations specified by the 'out' arglist. \textsf{ObjReturn} then returns to \( O_1 \), via the trap handling code.

If \( O_2 \) is not found locally when thread \( t \) enters the kernel during object call, \( O_2 \) is assumed to be remote and an invocation request is broadcast to other nodes. The RPC server on the node that has \( O_2 \) acknowledges the invocation request and creates a local slave process to invoke \( O_2 \) on behalf of \( t \). The slave process proceeds to invoke \( O_2 \) locally, and when the invocation completes, it sends the return arglist back to the invoking node. When the return arglist is received, it is acknowledged, and \textsf{ObjReturn} is called to return to \( O_1 \). The two important differences between local and remote invocation are the necessity to send messages through the local area network, and the need to employ a slave process at the remote node to perform the invocation.

**Measurements and improvements**

We added a profiling facility to the Clouds kernel. The mechanism has two components:

1. The program counter (PC) is sampled every clock tick (10ms). Each time the PC is sampled, a counter corresponding to the value of the PC is incremented. Each 4 bytes of kernel code has a corresponding counter, and a separate counter is incremented when the value of the PC falls in user space.
2. For each procedure call, the caller is noted, plus a count is maintained of the number of times the procedure has been called by this caller.

The PC sampling and the procedure call counts are collected during run-time. This information is then used to construct a graph of the dynamic call sequence. The dynamic call graph along with the symbol table and the static graph of the program are then used to produce a detailed analysis of the kernel during the time when profiling
is enabled. For each procedure, the information produced includes the percentage of
the total run time spent in it, the number of times it is called, and the percentage of
the time spent in each of its children. The algorithms used to build the dynamic and
static graphs, and to calculate the time spent in each procedure are based on the Unix
utility gprof.

In the following experiments, the objects involved are brought in memory to factor
out the secondary storage access, and the profiling is done on an otherwise idle system.
The costs of occasional reception of broadcast messages from the Ethernet plus the
cost of the profiling code itself are factored out of the results.

Local invocation

To measure the costs involved in object invocation, two objects Caller and Callee are
used in the following experiments. In each experiment, five separate runs are made,
with object Caller calling an entry point in object Callee 10,000 times, transferring-in
0 to 4096 bytes of data. Object Callee transfers-out 0 to 2048 bytes. Table I lists the
cost of each object invocation—the sum of kernel and user times. Tables II and III

Table I. Local object invocation costs

<table>
<thead>
<tr>
<th>Bytes (in + out)</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.4910</td>
</tr>
<tr>
<td>32 + 16</td>
<td>3.6040</td>
</tr>
<tr>
<td>64 + 32</td>
<td>3.6430</td>
</tr>
<tr>
<td>128 + 64</td>
<td>3.6710</td>
</tr>
<tr>
<td>256 + 128</td>
<td>3.7670</td>
</tr>
<tr>
<td>512 + 256</td>
<td>3.8770</td>
</tr>
<tr>
<td>1024 + 512</td>
<td>4.3020</td>
</tr>
<tr>
<td>2048 + 1024</td>
<td>5.1580</td>
</tr>
<tr>
<td>4096 + 2048</td>
<td>6.8220</td>
</tr>
</tbody>
</table>

Table II. Local object invocation, 0 bytes

<table>
<thead>
<tr>
<th>Time, ms</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total kernel mode time</td>
<td>2671</td>
</tr>
<tr>
<td>Trap handling</td>
<td>160</td>
</tr>
</tbody>
</table>
| ObjInvoke
  Locating object in memory | 97 | 3.6 |
  Checking rights + context switch + misc | 265 | 9.9 |
  Processing args | 99 | 3.7 |
  Allocating the ObjRec structure | 769 | 28.8 |
  Disabling and enabling interrupts | 287 | 10.7 |
| ObjReturn
  Context switch + misc | 215 | 8.0 |
  Processing Args | 176 | 6.6 |
  Deallocationing ObjRec structure | 285 | 10.7 |
  Disabling and enabling interrupts | 287 | 10.7 |
  Other | 31 | 1.2 |
| User mode time | 820 |   |
Table III. Local object invocation, 1024 bytes

<table>
<thead>
<tr>
<th>Time, ms</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3374</td>
<td>100.0</td>
</tr>
<tr>
<td>160</td>
<td>4.7</td>
</tr>
<tr>
<td>97</td>
<td>2.9</td>
</tr>
<tr>
<td>265</td>
<td>7.9</td>
</tr>
<tr>
<td>463</td>
<td>13.7</td>
</tr>
<tr>
<td>760</td>
<td>22.5</td>
</tr>
<tr>
<td>297</td>
<td>8.8</td>
</tr>
<tr>
<td>928</td>
<td>28.7</td>
</tr>
</tbody>
</table>

Table IV shows the cost of object invocation after these two modifications to the kernel. Note that the times shown in the table are the average cost per invocation, and include kernel and user mode times. Tables V and VI show the corresponding breakdown of the object invocation time for the two cases of 0 and 1024 bytes of transferred-in data,
### HARDWARE SUPPORT FOR OBJECT INVOCATION

#### Table IV. Modified local object invocation costs

<table>
<thead>
<tr>
<th>Bytes (in + out)</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.2610</td>
</tr>
<tr>
<td>32 + 16</td>
<td>2.2490</td>
</tr>
<tr>
<td>64 + 32</td>
<td>2.2900</td>
</tr>
<tr>
<td>128 + 64</td>
<td>2.3240</td>
</tr>
<tr>
<td>256 + 128</td>
<td>2.4040</td>
</tr>
<tr>
<td>512 + 256</td>
<td>2.5900</td>
</tr>
<tr>
<td>1024 + 512</td>
<td>3.0220</td>
</tr>
<tr>
<td>2048 + 1024</td>
<td>3.7190</td>
</tr>
<tr>
<td>4096 + 2048</td>
<td>5.5950</td>
</tr>
</tbody>
</table>

#### Table V. Modified local object invocation, 0 bytes

<table>
<thead>
<tr>
<th></th>
<th>Time, ms</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total kernel mode time</td>
<td>1441</td>
<td>100.0</td>
</tr>
<tr>
<td>Trap handling</td>
<td>160</td>
<td>11.1</td>
</tr>
<tr>
<td>ObjInvoke</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Locating object in memory</td>
<td>97</td>
<td>6.7</td>
</tr>
<tr>
<td>Checking rights + context switch + misc</td>
<td>265</td>
<td>18.4</td>
</tr>
<tr>
<td>Processing args</td>
<td>99</td>
<td>6.9</td>
</tr>
<tr>
<td>Allocating the ObjRec structure</td>
<td>199</td>
<td>13.8</td>
</tr>
<tr>
<td>ObjReturn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context switch + misc</td>
<td>215</td>
<td>14.9</td>
</tr>
<tr>
<td>Processing args</td>
<td>176</td>
<td>12.2</td>
</tr>
<tr>
<td>Deallocating ObjRec structure</td>
<td>199</td>
<td>13.8</td>
</tr>
<tr>
<td>Other</td>
<td>31</td>
<td>2.15</td>
</tr>
<tr>
<td>User mode time</td>
<td>820</td>
<td></td>
</tr>
</tbody>
</table>

#### Table VI. Modified local object invocation, 1024 bytes

<table>
<thead>
<tr>
<th></th>
<th>Time, ms</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total kernel mode time</td>
<td>2094</td>
<td>100.0</td>
</tr>
<tr>
<td>Trap handling</td>
<td>160</td>
<td>7.6</td>
</tr>
<tr>
<td>ObjInvoke</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Locating object in memory</td>
<td>97</td>
<td>4.6</td>
</tr>
<tr>
<td>Checking rights + context switch + misc</td>
<td>265</td>
<td>12.7</td>
</tr>
<tr>
<td>Processing args</td>
<td>463</td>
<td>22.0</td>
</tr>
<tr>
<td>Allocating the ObjRec structure</td>
<td>182</td>
<td>8.7</td>
</tr>
<tr>
<td>ObjReturn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context switch + misc</td>
<td>224</td>
<td>10.7</td>
</tr>
<tr>
<td>Processing args</td>
<td>477</td>
<td>22.8</td>
</tr>
<tr>
<td>Deallocating ObjRec structure</td>
<td>182</td>
<td>8.7</td>
</tr>
<tr>
<td>Other</td>
<td>44</td>
<td>2.1</td>
</tr>
<tr>
<td>User mode time</td>
<td>928</td>
<td></td>
</tr>
</tbody>
</table>
respectively. The time to enable/disable interrupts is now an insignificant portion of the object invocation time and is included as part of allocating the ObjRec structure in Tables V and VI. Comparing Tables II and V, the total kernel mode time spent on object invocation is reduced by 45 per cent from an average of 2671 μs to 1441 μs.

Though more tuning seems possible, the decompositions of costs in Tables V and VI are representative of a typical object invocation mechanism implemented on a conventional machine. Some of the costs shown in the Tables cannot be eliminated. For example, trap handling, checking rights, and initializing the ObjRec structure consist of straight-line code that cannot be eliminated. However, the measurements point to a mismatch between the invocation mechanism and the machine architecture. This mismatch is discussed next.

**TLB flushing costs**

As mentioned before, two context-switch operations are needed per object invocation. Each context-switch operation costs around 10-18 per cent of kernel mode object invocation time (see Tables V and VI). Most of this cost is eliminated using a protected procedure call to effect object invocation (see next section). However, there is still a hidden cost that results from switching address spaces on invocation. This hidden cost is due to flushing the TLB, and it manifests itself mostly in user mode time. We now investigate this point in more detail.

Conventional memory management units (MMU) provide a process with one virtual address space. These MMUs typically have the following three components:

1. An address-space pointer that points to the address-space mapping table, also referred to as the page table. Address-space switching is usually accomplished with a hardware pointer update in most MMUs.
2. A page table that holds mapping and protection information. In most MMUs the page tables are kept in physical memory. The page table may be organized as a linear table (e.g. Vax-11), a tree (e.g. Motorola 68851, Sun-3 MMU) or a hash table (e.g. IBM PC/RT, HP Precision Architecture). The latter two organizations are aimed at minimizing the amount of physical memory space that has to be reserved for holding the page tables when the virtual memory space is large and sparse. Organizations as a linear table necessitates the use of other mechanisms, such as paging the page tables themselves, to limit the size of the page tables in physical memory.
3. A TLB that holds recent virtual to physical address translations. A TLB is a high-speed associative cache and its purpose is to eliminate the need to access page tables that are often kept in main memory on each virtual-to-physical translation. In many MMUs, TLB entries are invalidated on every address space switch.

Object-based systems such as Clouds present a problem that is not very well handled by conventional MMUs. A thread in the course of its computation traverses the virtual address spaces of several objects. The operating system assigns a unique address space for each object. When a thread invokes an entry point in an object two events occur:

1. The memory management hardware switches the address space for the currently executing thread such that memory accesses henceforth will use the address space of the invoked object.
2. The TLB is flushed due to the switch in address spaces.

When a thread returns to the caller object, these two steps are repeated to switch back to the address space of the caller object.

It is known that the frequency of context switches can affect the TLB performance significantly. In conventional systems such as Unix, if the TLB size is large enough, adequate hit-rate performance can be achieved by choosing an appropriate organization of the TLB. However, our simulation studies show that in object-based systems increasing the TLB beyond a certain size does not help in reducing the hit rate.

To illustrate the effects of flushing the TLB on each object call and return, an experiment is conducted on a modified version of the Clouds kernel. An object, named Same, with two entry points SameCaller and SameCallee is used. SameCaller calls SameCallee, accesses $n$ pages of memory and repeats this sequence 10,000 times. SameCallee accesses $n$ different pages of memory and returns to its caller. Each page in memory is 512 bytes in size. The Clouds kernel is modified such that an intra-object invocation does not result in a context switch. All the kernel steps involved in object invocation are performed, except that the actual context switch instructions are replaced with no-op instructions. Figure 2 compares the average costs of object invocation with and without TLB flushing. In the figure, $t_T$ refers to the average total object invocation time, and $t_u$ refers to the average time spent in user mode per object invocation. The total number of pages accessed in SameCaller or SameCallee per invocation is more than $n$ because a number of code and stack pages are additionally accessed during each invocation.

On a context switch, the process TLB is flushed but the kernel TLB is left intact. After the context switch, almost all accesses to the process space (i.e. to thread data
and to the newly installed object) are made in user mode. Therefore, the cost of TLB flushing manifests itself mostly in user mode time $t_u$. There are, however, few pages in the process space that are accessed by the kernel after the context switch and before starting execution in the newly mapped object. On object call, some parameters (around 50 bytes) are pushed on the user stack (thus accessing at most two pages), and on object return the output parameters are copied into the user object. Therefore, TLB flushing affects mostly user mode time.

In Figure 2, comparing $t_{T,\text{flush}}$ to $t_{T,\text{no flush}}$ for the case of $n = 2$, shows that $t_{T,\text{no flush}}$ is 17.9 per cent smaller than $T_{T,\text{no flush}}$. For $n = 5$, the difference is 19.4 per cent. In the next section, we discuss a TLB design that supports caching of recent address translations across object invocations.

**Remote invocation**

To measure the cost of remote object invocation, we use the Caller and Callee objects again, but in a distributed setting, with each object loaded on a different machine. In these experiments, no data is transferred back from Callee to Caller. Caller invokes an entry point in Callee, the processor on which Caller executes remains idle until the invocation returns, and the invocation is repeated 1000 times. Column R1 in Table VII shows the total cost per object invocation for a range of transferred data. Tables VIII and IX provide a rough breakdown of the costs after eliminating CPU idle time. The following observations can be made based on these measurements:

(a) As mentioned before, a slave process is created on the remote node to perform the invocation locally on behalf of the invoking thread. Table VIII shows that around 43 per cent of the remote node CPU time is used in reclaiming slave processes and returning them to the pool of available slaves. Though no other activity is running on the machines during the experiment, the CPU idles only 23 per cent of the total experiment time.

(b) The general mechanism of the heap is used by the RPC and networking code to allocate memory buffers. We replaced the kernel heap with local free lists of memory blocks of the required sizes. We also modified the enable/disable routines

<table>
<thead>
<tr>
<th>Bytes (in)</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52.69</td>
<td>29.90</td>
<td>21.90</td>
</tr>
<tr>
<td>32</td>
<td>52.71</td>
<td>27.20</td>
<td>22.10</td>
</tr>
<tr>
<td>64</td>
<td>52.74</td>
<td>27.40</td>
<td>22.60</td>
</tr>
<tr>
<td>128</td>
<td>52.78</td>
<td>28.00</td>
<td>23.00</td>
</tr>
<tr>
<td>256</td>
<td>53.00</td>
<td>28.00</td>
<td>23.10</td>
</tr>
<tr>
<td>512</td>
<td>54.52</td>
<td>29.00</td>
<td>24.10</td>
</tr>
<tr>
<td>1024</td>
<td>64.92</td>
<td>32.60</td>
<td>27.50</td>
</tr>
<tr>
<td>2048</td>
<td>85.08</td>
<td>57.50</td>
<td>57.50</td>
</tr>
<tr>
<td>4096</td>
<td>125.15</td>
<td>89.90</td>
<td>89.90</td>
</tr>
</tbody>
</table>
as previously described in local invocation. These modifications are included in the following two experiments, shown in columns R2 and R3 in Table VII, and they reduce remote invocation time by approximately 9 per cent.

To illustrate how slave reclamation affects system behaviour, the same remote invocation experiment is repeated without reclaiming slaves, i.e. assuming that process reclamation costs zero time. The results are shown in column R2 in Table VII. The objective is to illustrate the effect of slave reclamation on RPC cost. Note that the difference between R1 and R2, when transferring 1024 bytes, is 27.02 ms. Of this time, 21.31 ms are due to slave reclamation costs and the rest are due to the modifications to buffer allocation and the enable/disable routines. The results of eliminating the two acknowledgements from the RPC protocol are shown in column R3 of Table VII.
From these experiments, the following observations are made concerning remote object invocation:

(a) Slave start/reclamation is very important and must be done fast. In the R1 case above, reclaiming slave processes is a bottle-neck that slows both the local and remote nodes. While servicing remote invocation requests, the Clouds kernel is also trying to prepare the 'dead' slaves for more work. Though the Clouds implementation of slave reclamation could be improved, there is always some cost for assigning cohort processes to perform remote invocations.

(b) The Clouds kernel passes little additional data on an object invocation, though a sophisticated operating system is expected to pass more thread-specific data. This information may include the thread identifier name, accounting information, controlling terminal information, etc. The operating system would then use this thread-specific information to create an environment similar to the invoking thread's environment. Passing additional data and creating a new thread environment on the remote node is expected to add more time to remote procedure calls.

Thus, the additional cost in a remote (non-local) invocation involves (a) sending messages across the network, (b) setting up an environment similar to the invoking thread's environment, and (c) assigning a cohort process to do the actual work. An alternative to sending the computation to a remote node is to bring the required data to the local node. This alternative, which we call distributed shared memory, has been explored in other systems, such as Emerald and Apollo Domain. In a subsequent section, the mechanisms needed to support distributed shared memory are presented, and a hardware module to assist in this function is proposed.

MEMORY MANAGEMENT SUPPORT

Based on our measurements, we believe that object-based systems would benefit considerably if the machine architecture were to provide support for a process to traverse and cache recent address translations in multiple address spaces. In many MMUs, a change in address-space pointer also results in flushing the TLB. For example, the Vax load process context instruction changes the page table pointers of process space and flushes the process space half of the TLB. Some MMUs, such as the Motorola 68851, the Amdahl 470V/7 and the IBM 3033, associate a virtual address tag with the TLB entries, allowing entries for more than one process to be in the TLB at the same time. In the Motorola 68851, a process identifier is stored in the TLB as part of the tag. Therefore, apart from the usual replacement of entries when the TLB is full, only when a process identifier is re-used the entries corresponding to that process identifier need to be flushed from the TLB. However, these MMUs do not meet the requirements of object-based systems, as will be evident from the discussion at the end of this section.

To better support object invocation, a scheme is presented in which the address space switch is effected by a protected procedure call mechanism, and virtual-to-physical address translations are cached across object invocations. This scheme, shown in Figure 3, is an extension of the one used in the Motorola 68851. The machine virtual address space is partitioned into three regions: the K space or kernel space, the P space or process space, where the currently running process has its stack and other
HARDWARE SUPPORT FOR OBJECT INVOCATION

Ptr to K-region: KBR
Ptr to P-region: PBR
Ptr to O-region: OBR

Index into OT: COR

Location in O-region: VPN, PPN
Location in K-region: VPN, PPN

A return from an object invocation is similar, with the operating system setting the OBR to the address space of the caller object. As is evident from this description, with our MMU the invocation is effected by a protected procedure call instead of a context switch.

The TLB entries are tagged with (object identifier, VPN) for O space translations. Translations in the P and K spaces are cached in the TLB with (C_P, VPN) and (C_K, VPN) as the tags respectively, where C_P and C_K are MMU constants outside the range of OT indices. The P space entries (with C_P tag) are flushed on every process switch.

Virtual addresses generated by the CPU are translated as follows: an address from the O space is prefixed with the contents of COR, and an associative search of the TLB is performed. A hit occurs when the (COR, VPN) pair matches the TLB tag. An address from the P or K space is prefixed with the appropriate MMU constant, either C_P or C_K, for TLB look-up. A hit occurs when the (Const, VPN) pair matches the TLB tag.
When an address-space pointer is re-used (e.g., when an object is destroyed), system software instructs the MMU to flush any matching entry in the OT with the same contents as the address space pointer. The MMU, in turn, flushes any corresponding entries in the TLB.

**Evaluation**

We evaluated our scheme using trace-driven simulation, and compared its performance to the usual case of flushing the TLB on each object call and return (see Reference 20 for more details). In performing this study, we were faced with the lack of traces of programs running on object-based systems. The current Clouds implementation runs only a few ‘toy’ programs, and does not support instruction tracing.

The following approach was used to generate the input for our simulator. A number of programs written in the C programming language, each of which consisted of several separately compiled modules, were executed under Unix on a Vax-11, and an address trace was generated for each program. The address traces were in turn used to drive a simulator that treated inter-module calls as object calls. Whenever a call (return) crossed from one module to another, it was treated as an object call (return). Intra-module calls were treated as normal procedure calls. Calls to library routines, as well as data accessed while executing inside an object were treated as part of that object’s virtual address space.

Modular programs tend to ‘group related functions and data in the same module, and to pass data by value across modules. This behaviour is very similar to what we expect to have on Clouds, with separate objects replacing program modules. The test programs used in the simulation follow this style of programming, with an average size of 500 lines of code per module, which is about the size of a Clouds object. We, therefore, believe that traces generated as explained above are appropriate for evaluating our scheme. The traces, however, lack kernel mode addresses, and therefore our simulation does not consider translation of K space addresses.

The results of our performance study are summarized below, where T1 refers to our scheme, and T0 refers to the usual case of flushing the TLB on object invocations:

1. For a given TLB size, the miss ratio of T1 is consistently less than the miss ratio T0. For a TLB size of 128, the miss ratio of T1 is 100 times less than the miss ratio of T0.
2. The T1 miss ratio decreases as the TLB size increases beyond the point when the T0 scheme miss ratio levels off. T1 uses the additional TLB entries to cache more address translations that may be used in the future, unlike T0 which flushes cached translations on each object call and return.
3. Depending on ratio $T_h / T_m$ (where $T_h$ is the hit cycle time and $T_m$ is the miss cycle time), the overall percentage improvement of T1 over T0 ranges from around 15 to 30 per cent.

The memory management unit that we presented is tailored to support object invocation. It is an engineering solution combining the features available in commercial MMUs. The strengths and weaknesses of commercial MMUs vis-à-vis the features required for an efficient implementation of Clouds are discussed below:

(a) The ability to cache TLB information across object invocation. The cost of
flushing TLB entries at object call/return is an implicit cost that affects mostly user mode time. Systems with one large virtual address space do not require flushing the TLB across object invocations. Examples of such systems include the IBM RT/PC, HP Precision Architecture and SPUR machines.  

(b) *The ability to implement an object call/return by performing a small number of operations on the MMU.* The cost of required MMU operations is an explicit part of object invocation. This cost is quantified by the number of MMU registers, either in software or hardware, that need to be modified on each object call/return. In our proposed MMU, changing the OBR is the only operation that is required to effect an object call/return. A paper design of object invocation implementation on segmented single virtual address space systems, such as IBM RT/PC and HP Precision Architecture, reveals that only a small number of MMU operations is required. However, our experience in implementing the new Clouds kernel on the Sun-3 MMU reveals that many MMU operations are needed on this machine.  

(c) *The ability to represent sparse address spaces efficiently.* Inefficient representation of sparse address spaces results in page tables that have large memory requirements and lengthy initialization time, relative to the size of address space actually allocated. As mentioned before, page tables organized as trees or inverted tables are suitable for representing sparse address spaces, whereas linear page tables are not.  

(d) *The ability to share memory among address spaces.* This feature is required to support sharing of segments among virtual spaces. Sharing segments in our proposed scheme and other MMUs with page-tables organized as trees is easy. A segment is shared between virtual spaces by sharing a page table subtree between the different page tables. MMUs with inverted page tables (e.g. IBM RT/PC) and virtually-addressed data caches (e.g. SPUR) cannot have two different virtual addresses mapping into the same physical address (i.e. no aliasing). Sharing of memory between different spaces is possible only through hardware segment sharing, which imposes limitations on the number of segments per object and the way they can be shared between virtual spaces. Clouds advocates a model of programming with a large number of possibly small-sized segments. Any segment is potentially sharable among virtual spaces. Because sharing can only be at the hardware segment level, a hardware segment maps exactly one software segment. For example, in the IBM RT/PC the CPU generates 32-bit virtual addresses with the high-order four bits of the virtual address selecting one of sixteen segment registers. In a paper design of object space implementation on the IBM PC/RT, the sixteen segments are allocated as follows: seven each for O and P spaces and two for K space. Each hardware segment maps at most one software segment.  

The HP Precision Architecture organizes its virtual space differently. Each virtual address is composed of a segment register and an offset. User software can change some of the segment registers without kernel assistance. Using such an organization, the number of software segments per virtual space is not constrained by the number of hardware registers. However, user software is responsible for loading the segment registers with hardware segment numbers before accessing a segment that is not already mapped. This organization complicates user software and may result in performance degradation if the object is
composed of many segments. Moreover, the segment registers are loaded with hardware segment numbers that the kernel has to set up. The kernel has to communicate these number to user software when objects are loaded in memory, and has to make sure that no object still uses a hardware segment number that is about to be reused.

(e) The ability to allocate/deallocate ranges of addresses easily. The ability to allocate/deallocate ranges of addresses is related to sharing of memory segments among spaces, because ranges of addresses are allocated/deallocated at the segment level. The number of page table and MMU operations required to allocate/deallocate a range of addresses affects the cost of attaching and detaching segments to virtual spaces. Some systems (e.g. Vax and Sun-3 MMU) require traversing a potentially large number of page-table entries to allocate or deallocate a range of addresses. Both IBM RT/PC and HP Precision Architecture adequately support this feature.

In summary, MMUs with a single large virtual space come closest to meeting the requirements of maintaining an object space. They do not require TLB flushing across object invocation, and may not require a large number of operations to implement object call/return. However, such MMUs either restrict the amount of sharing possible between virtual spaces, or introduce software complexities when sharing segments.

The requirements discussed so far are basic to managing object space. Systems such as Clouds and Argus support the notions of atomicity of computation and recoverability of data, sometimes referred to as transactions. This notion requires that memory segments be recoverable if a computation needs to be aborted. Software techniques such as shadowing and logging are usually employed to implement transactions. However, it is possible to reduce the burden on the software by providing some mechanisms in the MMU for supporting transactions. IBM RT/PC implements one such mechanism called transaction locking in its MMU. In Reference 27, Chang and Mergen report on implementing a transaction system using these mechanisms. We are currently evaluating the efficacy of incorporating transaction support in our MMU design. It should be noted that such a mechanism can be easily added to our present design.

DISTRIBUTED SHARED MEMORY

In a distributed object-based system, the virtual address spaces of all objects can be viewed as constituting a global distributed shared memory. Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-mapping mechanism. Local object invocation involves mapping the required memory segments of the object into the address space of the invoking process by installing the object as the current O space with the process's P space. The current trend in structuring distributed systems is to use a collection of diskless computational servers or workstations, and a few data servers or file servers. In such an environment, the code and data for the local invocation has to be paged-in from the data server. Further, for remote object invocation we have one of two choices: the first choice is to perform the computation at the node where the object resides through remote procedure call. The second choice is to make the invocation appear local by bringing in the segments required for the invocation. Whereas the former has to be
HARDWARE SUPPORT FOR OBJECT INVOCATION

supported for immovable objects such as an object that reads disk blocks, the latter may be a better choice for movable objects. There are two reasons to support this belief:

(a) the principle of locality\(^8\) that suggests an invocation or other invocations in the same object may be repeated.

(b) the reduction in computational overhead due to the elimination of slave processes management to support remote invocation at the node where the object resides.\(^21\)

In References 29 and 21 the concept of distributed shared memory as an alternative to RPC is proposed, the algorithms to maintain strong and weak memory consistency are presented, and the algorithms are evaluated using simulation. Each object is owned by one node, and segments of an object can be at other nodes temporarily. Each node has associated with it a ‘network cache’—a repository for recently accessed remote memory segments and their owners (nodes). A node that caches a segment from a remote owner is called a keeper of the segment.

The organization (Figure 4) proposed inside each node of the network is the following: a host that executes distributed applications; a distributed shared memory controller (DSMC) together with the network interface assists the host in mapping memory segments, both local and remote, into the virtual address spaces of the application processes. There is a minimal kernel on the host that traps system calls and virtual address translation faults. The DSMC is also in control of the network. The system memory logically is partitioned into two parts: one part, object memory, is for housing the segments of locally created objects; the other part, network cache, is for caching segments from remote objects. Conceptually there are two lists of process control blocks: the hostlist that the host looks at to schedule runnable processes and the DSMC list that the DSMC looks at to service memory segment requests. The host enqueues processes that fault on virtual address translation in the DSMC list. The DSMC enqueues processes that have become runnable again after the fault service in the host list.

The basic operations provided by the DSMC are get and discard. The get operation is used to fetch a segment (or a part thereof) from its owner, whereas discard is used to return a segment to its owner. The DSMC provides synchronization primitives as separate P and V semaphore operations, or as combined access and lock operations using the get and discard primitives.

---

**Figure 4. Hardware implementation**
Using the get primitive a segment may be acquired in one of four modes: read-only, read-write, weak-read or none. Read-only mode signifies non-exclusive access but guarantees that the segment will not change until the node explicitly discards the segment. Read-write mode signifies exclusive access (for the node) with a guarantee that the segment will not be thrown away until the node explicitly discards the segment. Weak-read mode signifies non-exclusive access with no guarantee whether the segment will change or not. None mode signifies exclusive access with no guarantee whether the segment will be thrown away or not.

We evaluated our DSMC scheme and compared it to RPC using simulation. In the simulation, the DSMC is treated as a software module that is part of the kernel. The performance study shows that

1. Over a range of object invocation locality, DSMC has significant advantages over RPC.
2. Use of distributed shared memory achieves automatic distribution of processing load, by performing the invocation locally instead of on the owner nodes.
3. Instead of addressing memory coherency and synchronization separately, applications that can be modelled as readers/writers problems benefit considerably when locking and segment access primitives are combined.

In Reference 5 the design and implementation of a native kernel that integrates distributed shared memory for network-wide memory management is discussed. The DSMC primitives are provided to the operating system as a set of mechanisms for managing memory in the network. Policy decisions, such as when to use the RPC mechanism and when to use the DSMC primitives, are in the operating system. For example, during periods of high concurrent access to an object, the operating system uses the DSMC primitives to locate the object at one node and uses RPCs to invoke the object, thereby reducing data movement across the network.

Our work is built on the large body of work that exists in maintaining cache coherence in multiprocessors. The DSMC algorithms deal with providing the mechanisms needed to maintain consistency of shared data in a non-shared memory architecture. Kai Li addresses the same problem, wherein the entire memory is composed of untyped pages that are potentially sharable for both reads and writes. The novelty in our work is in exploiting application semantics to type the segments as read-only or read-write, and using the type specifiers in the DSMC primitives as hints for simplifying consistency maintenance. By merging process synchronization with data transfer, the DSMC primitives provide mutual exclusion for free.

Other researchers have proposed the use of shared memory as a distributed systems structuring concept. The Apollo Domain system is a loosely coupled network of computers that provide the user with a view of a single-level store. This view allows programs to share files, specifying the semantics of sharing, such as exclusive/non-exclusive, at the time of opening the file. On opening the file, it is mapped into the virtual address space of the program, and henceforth reads and writes to the file are no different from simple memory reads and writes. These system architectures assume the existence underneath of a consistency preserving mechanism similar to Li's and ours. Distributed shared memory in an object-based environment plays a role similar to the one played by network file systems, such as NFS and Sprite, in conventional systems.
HARDWARE SUPPORT FOR OBJECT INVOCATION

CONCLUSIONS AND FUTURE WORK

The Clouds kernel served well as a prototype, and enabled us to gain important insight into the requirements of the object-based model of computation. However, it has several drawbacks. It is a monolithic kernel that is hard to modify and maintain, and the implementation is very machine-dependent. In addition, message transmission in Clouds is slow. Sending messages from one process running in a local kernel to a process running in a remote kernel takes roughly 10 ms. Most of this cost can be attributed to a poor network interface. Other systems, such as QuickSilver and V, report numbers in the range of 6 to 1.5 ms on faster hardware.

Our notions on structuring object-based operating systems has matured since the prototype design was begun. A new kernel for Clouds, called the Ra kernel, has been designed and implemented. Ra runs on the Sun-3 architecture, and incorporates support for distributed shared memory. The DSMC has been implemented in software and we are currently in the process of evaluating the implementation. Further refinement and implementation of an MNU tailored to object-based systems that incorporates our TLB scheme, and a hardware module that implements the DSMC protocol are some of the work we have identified for future research.

ACKNOWLEDGEMENT

This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.

REFERENCES

Hardware Support for Interprocess Communication

U. MAKISHORE RAMACHANDRAN, MARVIN SOLOMON, AND MARY K. VERNON, MEMBER, IEEE

Abstract—In recent years there has been increasing interest in message-based operating systems, particularly in distributed environments. Such systems consist of a small message-passing kernel supporting a collection of system server processes that provide such services as resource management, file service, and global communications. For such an architecture to be practical, it is essential that basic message exchanges be fast, since they often replace what would be a simple procedure call or "kernel call" in a more traditional system. Careful study of several operating systems shows that the limiting factor, especially for small messages, is typically not network bandwidth but processing overhead. Therefore, we propose using a special-purpose coprocessor to support operating systems. Our research has two parts. First, we partitioned an actual message-based operating system into computation and communication parts, executing, respectively, on a host and a message coprocessor interacting through shared queues, and measured its performance on a multiprocessor. Second, we designed hardware support in the form of a special-purpose smart shared memory and demonstrated the benefits of these components through analytical modeling using Generalized Timed Petri Nets. Our analysis shows good agreement with the experimental results and indicates that substantial benefits may be obtained from both the partitioning of the software between the host and the message coprocessor and the addition of a small amount of special-purpose hardware.

Index Terms—Architecture support, bus protocol, distributed systems, measurements, message-based operating systems, performance Petri nets, system architecture.

I. INTRODUCTION

We are interested in providing hardware support to improve the performance of distributed systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network (LAN). There are one or more processors and a certain amount of memory in each node. The nodes do not share memory; message exchange across the network is the only mechanism for communication between them. Many recent operating systems designs for such an environment [6], [25]–[27] place a message-passing kernel on each node, supporting processes and communication between them via explicit messages. This kernel supports both local communication—communication between processes on the same node—and nonlocal or remote communication (sometimes implemented via a distinguished network manager process).

Access to system services is requested via protected procedure calls in a traditional system, whereas in a message-based operating system it is requested via message passing. While a simple procedure call costs just a few instructions, and a protected procedure call (kernel call) costs a few hundred instructions, IPC costs a few thousand instructions in several systems that we studied [20]. Since message exchange is the basic kernel mechanism in message-based operating systems, the performance of the system depends crucially on the rate of message exchange. Our measurements (see Section II and [24]) as well as the measurements of others [6], [11] indicate that for small messages (which make up the vast majority of all messages sent [6]), the limiting factor is the high processing overhead that is incurred in message passing rather than limited network bandwidth or the time to copy messages from buffer to buffer.

There are two important figures of merit in this environment: roundtrip time, and message throughput. Roundtrip time is the elapsed time seen by an application between sending a message and receiving a reply from the intended receiving process. This figure of merit affects an individual application's performance. Message throughput is a global figure of merit that determines the performance of the entire system. Informally, it is the number of messages that the system handles per unit time. We show that a modest amount of additional hardware can significantly improve message throughput and average roundtrip time in a multiprogramming environment. We also show that additional hardware support in the form of high-level bus primitives affords even greater improvement in communication subsystem performance.

A. Background and Related Work

Available hardware support [1], [14] and previous modeling studies [12], [29] address the issue of off-loading communication protocols onto front-end processors, and provide evidence that this approach can have a significant performance payoff. However, these and other previous proposals of hardware support for interprocess communication (see survey in [21]) are more limited than the study reported in this paper in several respects.

First, previous work generally assumes "communication" to be the work that is performed by the operating system to satisfy nonlocal requests. However, for message-based operating systems, measurements by us (see Section II and [24]) and others [6] show that there is a high processing overhead for local communication as well. Existing hardware support for interprocess communication takes the form of

1) operations on structured data types in the instruction set architecture of the processor (often through microcode) such
send and receive in Intel's iAPX 432 [8]
2) network interfaces with direct access to host memory (e.g., Interlan Ethernet [14])
3) protocol processors that implement some form of transport level protocol for off-machine communication (e.g., TP Interface board from ABLE [1])
4) multiprocessor architectures with one processor performing message passing functions, generally for a group of processors (e.g., Kmap of Cm[10])
5) bus architectures that support higher level operations (e.g., VMEbus [9]).

Second, many proposals include only limited, low-level support for communication, leaving out support for operations such as address translation, control block manipulation, and kernel buffering, which account for a substantial portion of message passing overhead.

Finally, a front-end processor for a specific network protocol (such as TCP/IP [19]) may not mesh well with the operating system primitives, and therefore may incur higher overhead than necessary. The problem needs to be addressed at a higher level.

Proposed Organization

The organization inside each node is shown in Fig. 1. There is a host in each node that executes the message-based operating system and the applications. The shared bus, the shared memory, the message coprocessor, and the network interfaces together function as a single unit in assisting the host in message-passing activities. The host, the message coprocessor, and the network interfaces interact and synchronize via shared memory. This organization is similar to the ones studied in the network front-ends such as [12], and in commercial products such as the ABLE Easyway Port [1].

[7], the authors report on a similar hardware organization implementing the run-time support for an extended version of Hoare's CSP. However, what distinguishes our work from theirs is that we are not discovering coding inefficiencies in the system code that is executed on behalf of the operating system. Our solution to this problem suggests a system architecture that has a software aspect and a hardware aspect.

Our organization is aimed at improving the message throughput and reducing the roundtrip time by providing concurrent processing support for message passing. This organization raises two main questions.

1) How should the message-based operating system be partitioned between the host and the message coprocessor?

2) What kind of hardware assistance is appropriate to support interaction between the host, the message coprocessor, and network devices?

Our objectives in this research are a) to answer these questions, and b) to quantify the performance improvement due to the interaction between the host and the message coprocessor.

The rest of the paper is organized as follows. We summarize four of the distributed systems in Section II. The software partition that we implemented is discussed in Section III, and Section IV describes the hardware organization. Performance analysis is presented in Section V, and concluding remarks are given in Section VI.

II. MEASUREMENTS OF DISTRIBUTED SYSTEMS

We studied the design and implementation of four operating systems in detail: Charlotte [3], Jasmin [15], Quicksilver[26], and Unix. We profiled them (see [24] for more details) to ensure that we are not discovering coding inefficiencies in one operating system but see a trend that is common to all these systems. Our model of a distributed system assumes that processes communicate via explicit messages and that system services are provided by trusted server processes (as opposed to a monolithic kernel). Charlotte, Jasmin, and 925 belong to this model. However, all of these systems are experimental research projects. Therefore, we also studied Unix, which is not a message-based operating system, to see whether operating systems in extensive use suffer from similar problems.

To summarize the results from the studies, there are two important characteristics of distributed message-based operating systems.

Structure: System services in a distributed system are accomplished by a combination of computation and communication. By computation we mean processing done on behalf of the users. By communication we mean the system code that has to be executed to process a communication request.

Communication Overhead: There is a fixed overhead incurred in communication (independent of the message size) that can be decomposed into components such as checking the validity of an IPC call, addressing and manipulating control blocks, and short-term scheduling. There is a variable overhead due to kernel buffering that is dependent on the size of the message.
message and the number of times the message is copied from source to destination. This combined overhead is present for both local and nonlocal communication. For nonlocal communication there is additional overhead such as sending network packets, processing network interrupts, checksum calculation, and retransmission. For small messages, i.e., message size smaller than 100 bytes, copy-time is less than 20% of the total roundtrip time. For large messages, i.e., message size larger than 1000 bytes, copy-time begins to dominate the total roundtrip. Some systems (such as Accent [25] and Mach [2]) integrate virtual memory management with interprocess communication to reduce copy overhead for local message passing. However, given that copy-time accounts for less than 20% of the roundtrip time for short messages, communication overhead remains a significant performance bottleneck even in such systems.

III. SOFTWARE PARTITION

In a distributed system, users request system services by communicating with the servers. These servers compute to satisfy the requests possibly communicating with other servers. Through our profiling studies, we showed that a large percentage of the roundtrip time can be attributed to short-term process scheduling and control block manipulation functions. These kernel functions are performed for both local and nonlocal communication. Therefore, it is clear that message-passing support should be provided transparently for both local and nonlocal communication. Further, timing measurements for performing typical services on Unix (see [24] for more details) suggest that server computation times are comparable to communication times incurred in the message kernel. The structure of the distributed system and the results of our studies suggest the following partition of the message-based operating system between the host and the message coprocessor: computation on the host and communication on the message coprocessor (Fig. 1).

The servers execute on the host and the message-passing kernel executes on the message coprocessor. The code and private data structures of the message-passing kernel are in the local memory of the message coprocessor. The code and data structures for the servers reside in the local memory of the host. The shared memory contains the task control blocks and kernel buffers. The message coprocessor is in control of the network interface. There are two lists of task control blocks in the shared memory: the computation list and the communication list, representing work to do for the host and message coprocessor, respectively. The lists are ordered by task scheduling priority. The host interrupts and informs the message coprocessor that the communication list is nonempty. Similarly, the message coprocessor informs the host that the computation list is nonempty via interrupt. When the computation list is nonempty, the host gets the first task from the list and executes it until the task makes a communication request. At that point, the host enqueues the task on the communication list and starts executing the next task in the computation list. The task control block contains the information needed to process the communication request. When the communication list is nonempty, the message coprocessor gets the first task from the list and executes the communication processing code associated with that particular request. This processing will involve such chores as checking the validity of the IPC call, addressing and manipulating control blocks, kernel buffering, short-term scheduling decisions, sending network packets for nonlocal messages, and responding to network interrupts. As a result of this processing, a task that was waiting for a message may become ready to execute on the host. Network interrupts are serviced by the message coprocessor on a priority basis and lead to similar short-term scheduling decisions.

To verify the feasibility of such a software partition and to gather actual timing information, we implemented such a partition on an experimental system. 925 [26] is an early version of an experimental operating system (since renamed Quick-silver) for a network of workstations. For the purposes of this discussion, 925 is quite similar to the Stanford V Kernel [6]. The version we modified ran on a multiprocessor workstation with three Motorola 68000 processors [16], each with local memory, connected by a Versabus [17] to each other, a shared memory, and an early experimental version of the IBM token ring [5]. We emulated the message coprocessor with one of the processors, and measured the implementation to obtain the processing times for the kernel activities involved in message passing.

Through the implementation, we established the feasibility of partitioning the message-based operating system between the host and the message coprocessor. Another important fruit of this exercise was insight into the kinds of system data structures that are used in communication processing, the operations that are done on them, and the overhead for these operations. Buffers and lists of control blocks are the data structures that are extensively manipulated in communication processing. Operations on these data structures include copying and atomic queue manipulation. With the Motorola 68000 implementation it takes 220 µs of processing time to copy 40 bytes, and 74 µs of processing time to perform an atomic queueing operation (which involves obtaining a mutual exclusion lock, performing the queueing operation, and releasing the lock). There are four copy operations and 16 queueing operations (see [20] for details) in one roundtrip (nonlocal communication). Hence, these times are important since they constitute a significant portion of the total roundtrip time. Drawing on our implementation experience, we describe an improved hardware organization in the next section.

IV. HARDWARE ORGANIZATION

The system data structures in shared memory are manipulated by all the units inside each node. The operations on these data structures can be grouped into three categories: movement of blocks of data, queue manipulation, and simple read/write. The above groups of operations are general and applicable for implementing the semantics of interprocess communication of any operating system. Hence, it is appropriate to support these operations on the shared memory at the bus level.

2 The host to message coprocessor interaction results in an additional overhead for IPC, compared to a uniprocessor implementation of a message-based system. However, this overhead was less than 10% of the total roundtrip time.
Several recent bus proposals support block transfer primitives [4]. However, these bus proposals are intended for a versatile environment with multiple memory modules, processor modules, and device modules. In our environment, there is a limited shared memory holding task control blocks and kernel buffers. The units that access this memory are the message coprocessor, the host, and the network interfaces. This memory does not contain either “kernel programs” or “user programs.” On the contrary, it contains only protected kernel data structures that are manipulated by trusted kernel code executing in the message coprocessor and the host. Each unit that accesses this memory has exactly one outstanding request. In a limited controlled environment it would be more cost effective for the memory to handle multiplexed block transfers. Moreover, none of the existing bus proposals support atomic queue manipulation primitives.

We have designed a smart bus for message-passing support. To support the high-level primitives in this bus, we have signed a smart shared memory that implements these primitives. To put our bus design in the proper perspective, we would point out that the intent is not to invent a standard for future buses. In fact, we view the bus, the message coprocessor, the shared memory, and the network interfaces together as a single unit that provides message-passing support to the user at the level of the operating system primitives. This unit exists with the rest of the node architecture that includes a bus on which the host, the devices, and the host memory reside.

Smart Bus Overview

Smart bus connects the host, the message coprocessor, the network interfaces to the shared memory. Multiplexed block transfer and atomic queue manipulation are the transactions supported on the smart bus. Smart bus decouples requests for block transfers from the actual data transfers. The read memory caches information regarding block transfer requests (address and size) in an internal table, so that it can serve a lower priority request after servicing a higher priority one. The bus is never locked for arbitrary amounts of time, thus guaranteeing access for higher priority requests in finite time. A unit can have exactly one outstanding block transfer request. Therefore, the shared memory does not have to handle any flow control problems. Prioritized arbitration among competing units is supported on the bus. The arbitration scheme we use in our bus design is inspired by Futurebus but is simpler owing to the limited environment. Bus transfer rate is scalable with device technology due to the asynchronous protocol.

Physically, the bus includes 16 multiplexed address/data lines, four-lines for commands, and four-lines for a tag. In addition, there are arbitration lines for access control, protocol lines to complete the asynchronous handshake, and a reset line for startup. We refer to a transition on a clock edge as a clock-edge.

The number of multiplexed address/data lines in our design is 16, stemming from the fact that our experimental results were obtained from a 16-bit Versabus [17] implementation. To maintain compatibility with our experimental results we used 16-bit address/data lines. However, there is no inherent assumption in our design that would preclude extension to a wider bus.

The shared bus supports three categories of transactions: block requests, queue manipulation, simple read/write. Table I gives a summary of the smart bus commands and the time (measured in clock-edges) for performing the operations.

1) Block Requests: There are three transactions provided in this category: block transfer, block read data, and block write data. These primitives allow movement of blocks of contiguous data between the shared memory and other units in each node. They allow the shared memory to be multiplexed for handling simultaneous requests. Block transfer and block write data are initiated by the CPU’s and network devices. Henceforth, we refer to either a CPU or a network device as a processor. The processor that initiates block transfer specifies whether it is a read or a write. Block read data is initiated by the shared memory. While block transfer is the primitive used by the processor to convey the intent to the shared memory, block read data and block write data are the primitives used to effect the actual data movement.

In block transfer, the processor sends the starting address of the block and a count indicating the number of contiguous bytes of information to be transferred. The command (read or write) is specified on the command bus. Shared memory stores them in its internal table and responds by returning a tag that uniquely identifies the transaction. Block read data and block write data are primitives that are issued following the block transfer request. Both these primitives result in data transfer. Shared memory executes block read data to send the data along with the tag that uniquely identifies the processor of the block transfer request. The processor monitors the tag bus. When there is a tag match, the processor receives the data from the bus. Information transfer is in the opposite direction for block write data. Following a request to write a block of data, the processor executes block write data sending the data along with the tag to the shared memory. Shared memory receives them and uses the tag as an index into its internal table to get the address where the data are to be stored.

2) Queue Manipulation: There are three primitives provided in this category: enqueue control block, first control block, and dequeue control block. By presenting the memory as a singly-linked circular list of control blocks, these primitives allow atomic queueing operations to be performed on these lists. The data structure in memory looks as shown in Fig. 2. When presented with a list address, the memory unit
views it as the address of the location in memory ("List" in Fig. 2) that points to the tail of the list. Definitions of these primitives are given below. In each case, "list" refers to the location in memory that points to the tail (last element) of the list (Fig. 2).

1) **Enqueue(element, list)**: Add element to the tail of list and update list to point to the newly added item.

2) **First(list)**: Dequeue the first item in list and return a pointer to it. Return a distinguished value if the list is empty.

3) **Dequeue(element, list)**: Search for element in list and remove it. If element is not found, do nothing. This operation can be expensive if the list is long, but our experience with the 925 implementation shows that it is used only in exceptional circumstances.

3) **Read/Write**: In addition to the above transactions, the bus supports simple read/write primitives at byte granularity.

### B. Smart Shared Memory

We designed a bus architecture that is appropriate within the functional unit composed of the message coprocessor and the network interfaces. However, this design implicitly assumes that the shared memory has the necessary "intelligence" to handle the high-level requests of the smart bus. The design also assumes that the processors either themselves or within their bus interfaces have the necessary intelligence to generate these requests. Fortunately, even though the bus transactions are high-level, the nature of the environment makes these transactions feasible from the point of view of hardware implementation. Moreover, the nature of the environment makes it possible to provide these facilities at a reasonable "cost".

We demonstrate this feasibility through a detailed design of a *smart shared memory* [20]. The controller for the smart shared memory is microprogrammed, and has under 3000 bits of microcode. Based on the complexity of the design, we also show that the entire design can be packaged in two chips. The data path (without the memory system) can be implemented as a single chip with roughly 6000 active components. The sequencer can be implemented as a single chip with roughly 1000 active components.

### V. Performance Analysis

Our solution to the message-passing problem in distributed systems had two parts: *software partition* and *hardware organization*. While we implemented the software partition, limitations on the time and money we had available for fabricating and testing the hardware lead us to model rather than build the *smart bus* and the *smart shared memory*. Moreover, modeling allowed us to parameterize the design, thus enabling individual features to be evaluated. The results of our performance analysis were sufficiently encouraging that we are now considering an experimental implementation of the hardware.

We modeled our design using *Generalized Timed Petri Nets* (GTPN) [13], an extension of Petri nets [18] that allows assignment of firing durations to transitions and relative probabilities to alternative paths through the net. We then used a tool that builds the set of reachable states for the GTPN model and solves the resulting Markov chain to determine steady-state performance measures. Aggregate performance measures specified by the user (e.g., system throughput) are also computed automatically by the tool. This approach provides more precise results than simulation, but the formulation of the model requires some care lest the number of states become excessive. Some of the techniques we used to avoid state explosion are interesting in their own right. The interested reader is referred to [23] for details.

#### A. Architectures

We compare three architectures.

- **Architecture I** (Fig. 3) is a uniprocessor implementation of a distributed system. The message-based operating system executes on the host. The host is in control of the network interface.

- **Architecture II** (Fig. 4) is the organization we implemented in 925 (see Section III).

- **Architecture III** is similar to architecture II, with the difference that a smart bus interconnects the different units within each node and a smart memory serves as shared memory.

One important fruit of the implementation is that it gave us the timing values needed for driving the different models. These timing values are the processing times for the different components of message passing. In the architectures we are comparing, we assume the processors to be identical. Hence, the processing times we obtained from our implementation are applicable to all of them.

#### B. Workload Description

In this section, we describe the workload that we used as the basis for comparing the different architectures. While this is not the only possible workload, it is a typical workload in a distributed system. We plan to study other workloads in future.

A client loops making blocking *send* requests:

```
loop
    send;
end;
```

A server loops posting *receive* system calls:

```
loop
    receive;
    compute;
    reply;
end;
```

When the send and the receive match, a rendezvous takes place between the client and the server. The server then computes for a while processing the request in the message from the client. At the end of the computation phase, the server
C. Processing Times

In our implementation, the Motorola 68000 CPU was driven by an 8 MHz clock, yielding an instruction execution rate of roughly 0.3 MIPS [16]. Versabus [17] memory cycle time is on an average 1 µs. In our models, we assume an instruction execution time of 3 µs and a Versabus memory cycle time of 1 µs. We also assume that the four-edge handshake of smart bus equals Versabus memory cycle time and that the two-edge handshake equals half the Versabus memory cycle time. We should point out that a much higher speed is achievable for the smart bus with current technology. However, these conservative times for smart bus primitives give a more realistic basis for comparing the different architectures. Table II shows a comparison of implementing queue manipulation and block transfer operations for architectures II and III. For architecture II, each of enqueue, dequeue, and first involves the following steps to be performed by the message coprocessor: lock a semaphore, execute the queue manipulation algorithm (see Section IV-A2), and release the semaphore. The message coprocessor executes a program loop for reading or writing a block in architecture II. The processing time for this loop execution is shown in Table II. The message coprocessor in architecture III executes three instructions to initiate any of the smart bus primitives. For example, to initiate block transfer the message coprocessor writes the starting address, count, and command to its bus interface.

Tables III-VIII contain a breakdown of the communication time for one roundtrip conversation into component message-passing activities. The breakdown shows the processing time and the time spent in accessing shared data structures for both local and nonlocal conversations. The times for architecture II were obtained directly from our implementation. The times for architecture I were obtained from architecture II by eliminating the overhead for synchronization between the host and the message coprocessor. The times for architecture III were derived from architecture II after factoring in the primitives of the smart bus.

The following table shows the cumulative processing and shared memory access times for architectures II and III, and the percentage reduction in these times due to the smart bus primitives.

<table>
<thead>
<tr>
<th>Communication</th>
<th>Activity</th>
<th>Arch II</th>
<th>Arch III</th>
<th>Difference</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Processing</td>
<td>4770</td>
<td>3300</td>
<td>1470</td>
<td>30%</td>
</tr>
<tr>
<td></td>
<td>Shared Memory Access</td>
<td>828</td>
<td>612</td>
<td>216</td>
<td>26%</td>
</tr>
<tr>
<td>Nonlocal</td>
<td>Processing</td>
<td>6770</td>
<td>5200</td>
<td>1570</td>
<td>23%</td>
</tr>
<tr>
<td></td>
<td>Shared Memory Access</td>
<td>988</td>
<td>692</td>
<td>296</td>
<td>29%</td>
</tr>
</tbody>
</table>
TABLE II
Comparison of Processing Times

<table>
<thead>
<tr>
<th>Operation</th>
<th>Architecture II</th>
<th>Architecture III</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enqueue</td>
<td>60</td>
<td>14 9 1</td>
</tr>
<tr>
<td>Dequeue</td>
<td>60</td>
<td>14 9 1</td>
</tr>
<tr>
<td>First</td>
<td>60</td>
<td>14 9 2</td>
</tr>
<tr>
<td>Block</td>
<td>180</td>
<td>20 9 11</td>
</tr>
<tr>
<td>Block</td>
<td>180</td>
<td>20 9 11</td>
</tr>
<tr>
<td>Handshake</td>
<td>micro-seconds</td>
<td></td>
</tr>
<tr>
<td>Enqueue</td>
<td>Four-edge</td>
<td></td>
</tr>
<tr>
<td>Dequeue</td>
<td>Four-edge</td>
<td></td>
</tr>
<tr>
<td>First</td>
<td>Eight-edge</td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>One four-edge</td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>One four-edge</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>180</td>
<td>20 9 11</td>
</tr>
</tbody>
</table>

TABLE III
Architecture I: Local Conversation

<table>
<thead>
<tr>
<th>Processor</th>
<th>Initiator</th>
<th>Action</th>
<th>Time (microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Client</td>
<td>Syscall Send</td>
<td>1040 150</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Syscall Receive</td>
<td>650 120</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Match client with server</td>
<td>1240 140</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Compute Workload Parameter</td>
<td>1020 210</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Syscall Reply</td>
<td>140 60</td>
</tr>
<tr>
<td>Host</td>
<td>Restart Server</td>
<td>140 60</td>
<td></td>
</tr>
<tr>
<td>Host</td>
<td>Restart Client</td>
<td>140 60</td>
<td></td>
</tr>
</tbody>
</table>

TABLE IV
Architecture I: Nonlocal Conversation

<table>
<thead>
<tr>
<th>Processor</th>
<th>Initiator</th>
<th>Action</th>
<th>Time (microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Client</td>
<td>Syscall Send</td>
<td>1140 150</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Syscall Receive</td>
<td>200 30</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Syscall Receive</td>
<td>650 120</td>
</tr>
<tr>
<td>Host</td>
<td>Network interrupt</td>
<td>Match client with server</td>
<td>1790 210</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Compute Workload Parameter</td>
<td>1060 220</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Syscall Reply</td>
<td>1060 220</td>
</tr>
<tr>
<td>Host</td>
<td>Network interrupt</td>
<td>DMA in</td>
<td>200 30</td>
</tr>
<tr>
<td>Host</td>
<td>Network interrupt</td>
<td>Match client with server</td>
<td>1790 210</td>
</tr>
<tr>
<td>Host</td>
<td>Network interrupt</td>
<td>DMA in</td>
<td>200 30</td>
</tr>
<tr>
<td>Host</td>
<td>Network interrupt</td>
<td>Cleanup and Restart Client</td>
<td>830 130</td>
</tr>
</tbody>
</table>

D. Validation

Our experimental implementation on the 925 system differed from architecture II in two ways.
1) There were two hosts in each node instead of one.
2) The network interfaces required an additional copy from the kernel buffers to the memory-mapped network buffers in shared memory.

We used the workload described in Section V-B for performance measurements of the implementation. We validated a model for nonlocal conversations of our experimental implementation against these performance measures. Fig. 5(a), (b), and (c), shows the agreement between the experimental and model results: We note that for one and two conversations [Fig. 5(a)] the agreement is very good (within 3% for one and 10% for two). For three and four conversations [Fig. 5(b) and (c)], the model results are within 10% of the experimental results at high offered loads, while at low offered loads the deviation is within 25%. The optimistic prediction in the case of low offered load (high computation) is partly due to a load-leveling effect in the model not present in the experimental implementation: in the implementation, a process is bound to a particular host, whereas in the model, a request can be serviced on any available host. When the load is less communication-intensive, server processes spend a larger fraction of time on the host and as a result the throughput
TABLE VI
ARCHITECTURE II: NONLOCAL CONVERSATION

<table>
<thead>
<tr>
<th>Processor</th>
<th>Initiator</th>
<th>Action</th>
<th>Time (microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Processing</td>
</tr>
<tr>
<td>Host</td>
<td>Client</td>
<td>Syscall Send</td>
<td>320</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Syscall Receive</td>
<td>320</td>
</tr>
<tr>
<td>MP</td>
<td>Client</td>
<td>Process Send</td>
<td>1000</td>
</tr>
<tr>
<td>MP</td>
<td>Server</td>
<td>Process Receive</td>
<td>510</td>
</tr>
<tr>
<td>DMA</td>
<td>Client</td>
<td>DMA out</td>
<td>200</td>
</tr>
<tr>
<td>DMA</td>
<td>Server</td>
<td>DMA in</td>
<td>200</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Restart Server</td>
<td>60</td>
</tr>
<tr>
<td>MP</td>
<td>Server</td>
<td>Match client with server</td>
<td>1650</td>
</tr>
<tr>
<td>DMA</td>
<td>Server</td>
<td>DMA out</td>
<td>200</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Restart Server</td>
<td>60</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Syscall Reply</td>
<td>320</td>
</tr>
<tr>
<td>MP</td>
<td>Server</td>
<td>Process Reply</td>
<td>920</td>
</tr>
<tr>
<td>DMA</td>
<td>Server</td>
<td>DMA out</td>
<td>200</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Restart Server</td>
<td>60</td>
</tr>
<tr>
<td>DMA</td>
<td>Server</td>
<td>DMA in</td>
<td>200</td>
</tr>
<tr>
<td>MP</td>
<td>Server</td>
<td>Cleanup client</td>
<td>750</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Restart Client</td>
<td>60</td>
</tr>
</tbody>
</table>

Table VII
ARCHITECTURE III: LOCAL CONVERSATION

<table>
<thead>
<tr>
<th>Processor</th>
<th>Initiator</th>
<th>Action</th>
<th>Time (microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Processing</td>
</tr>
<tr>
<td>Host</td>
<td>Client</td>
<td>Syscall Send</td>
<td>220</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Syscall Receive</td>
<td>220</td>
</tr>
<tr>
<td>MP</td>
<td>Client</td>
<td>Process Send</td>
<td>612</td>
</tr>
<tr>
<td>MP</td>
<td>Server</td>
<td>Process Receive</td>
<td>451</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Match client with server</td>
<td>922</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Restart Server</td>
<td>60</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Syscall Reply</td>
<td>220</td>
</tr>
<tr>
<td>MP</td>
<td>Server</td>
<td>Process Reply</td>
<td>475</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Restart Server</td>
<td>60</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Host</td>
<td>Restart Server</td>
<td></td>
<td>60</td>
</tr>
</tbody>
</table>

Results
In this section, we present and compare the results of solving the models for the three architectures for the workload we described earlier.

1) Maximum Communication Load: Fig. 6(a) and (b) compares the throughput of architectures I, II, and III, under conditions of maximum communication load for local conversations and nonlocal conversations. The throughput results shown in Figs. 6 and 7 are based on a size of 40 bytes for each send or reply message. For architecture I, the throughput for local conversations is the same irrespective of the number of invocations, a fairly intuitive result. For architecture II, the throughput for one conversation is slightly less than that for predicted by the model is higher. However, despite this effect, the model results show good overall agreement with the experimental results.

Fig. 5. Model validation.
TABLE VIII
ARCHITECTURE III: NONLOCAL CONVERSATION

<table>
<thead>
<tr>
<th>Processor</th>
<th>Initiator</th>
<th>Action</th>
<th>Time (microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Processing</td>
</tr>
<tr>
<td>Host</td>
<td>Client</td>
<td>Syscall Send</td>
<td>220</td>
</tr>
<tr>
<td>MP</td>
<td>Client</td>
<td>Process Send</td>
<td>712</td>
</tr>
<tr>
<td>DMA</td>
<td>Client</td>
<td>DMA out</td>
<td>200</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Syscall Receive</td>
<td>220</td>
</tr>
<tr>
<td>MP</td>
<td>Server</td>
<td>Process Receive</td>
<td>451</td>
</tr>
<tr>
<td>DMA</td>
<td>Network</td>
<td>DMA in</td>
<td>200</td>
</tr>
<tr>
<td>MP</td>
<td>Network</td>
<td>Match client with server</td>
<td>1362</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Restart Server</td>
<td>60</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Compute</td>
<td>Workload Parameter</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Syscall Reply</td>
<td>220</td>
</tr>
<tr>
<td>MP</td>
<td>Server</td>
<td>Process Reply</td>
<td>573</td>
</tr>
<tr>
<td>DMA</td>
<td>Server</td>
<td>DMA out</td>
<td>200</td>
</tr>
<tr>
<td>Host</td>
<td>Server</td>
<td>Restart Server</td>
<td>60</td>
</tr>
<tr>
<td>DMA</td>
<td>Network</td>
<td>DMA in</td>
<td>200</td>
</tr>
<tr>
<td>MP</td>
<td>Network</td>
<td>Cleanup client</td>
<td>462</td>
</tr>
<tr>
<td>Host</td>
<td>Network</td>
<td>Restart Client</td>
<td>60</td>
</tr>
</tbody>
</table>

architecture I. The loss represents the overhead involved in the information transfer between the host and the message coprocessor. However, note that this loss is very small (=10%). Increase in throughput with the number of conversations is less than linear due to the finite bandwidth of the message coprocessor. Note that architecture III is significantly better than both architectures I and II. The smart bus reduces the overhead in communication processing by providing high-level bus transactions. These transactions are significantly faster than a software implementation (see Section V-C).

Fig. 6(b) illustrates the results for nonlocal conversations. The tendency to saturate with number of conversations is less pronounced for nonlocal conversations when compared to local conversations, since the processing load is spread across two nodes. Once again we note that architecture III performs significantly better than architectures I and II.

We note that architecture II does not do significantly better than architecture I (both local and nonlocal conversations). However, these graphs are for maximum communication load. Under these conditions the host is idle most of the time since there is no computation in any conversation. The premise behind partitioning the software is that load in a distributed system consists of a good mix of computation and communication. In the next section, we will discuss our results under such typical load.

2) Varying Workload: In this section, we compare architectures I, II, and III under the assumption that the server does a certain nonzero amount of computation before replying to the client. As we mentioned earlier (see Section V-B), offered load is defined as $L = C/(C + S)$, where $C$ is the communication processing requirement in one roundtrip and $S$ is the server computation time. $C$ is dependent on the architecture while $S$ is a workload parameter. Tables IX and X give the offered loads for different server-computation times in the three architectures for local and nonlocal conversations, respectively. Note that the offered load for a given server-computation time is the least for architecture III since it has the least $C$, and slightly higher for architecture II. The value of $S$ for a given service is the same for each of the three architectures. For example, our measurements of Unix on a processor that is about two to three times the speed of the modeled architecture show service times ranging from 0.2 to 6.1 ms (see [24]). Using Tables IX and X, we can read off the offered loads for each architecture given the server-computation time.

We want to be able to compare the performance of the three architectures for various servers. Fig. 7 illustrates how message throughput depends on offered load, as determined by
Since the offered load $L$ depends on $C$, which is a function of the particular architecture, we normalized the results by plotting throughput for each architecture as a function of the offered load a given server would produce on architecture I. Fig. 7(a) illustrates local conversations while Fig. 7(b) illustrates nonlocal conversations.

For architecture I, with local conversation, the results are independent of the number of conversations. Architecture II does slightly worse than architecture I for one conversation due to the overhead in passing information between the host and the message coprocessor. However, as the number of conversations is increased, the throughput improves considerably over architecture I. With a message coprocessor equal in processing speed to the host, the upper bound for throughput improvement (with no overhead between the host and the message coprocessor) is a factor of two. Architecture II approaches this limit over a range (0.5-0.9) of values for offered load. When the load is more computation intensive there is no significant gain in partitioning the software. The graph defines a region of operation of the distributed system in terms of mixture of computation and communication for which the message coprocessor is viable. By providing high-level bus primitives, architecture III does better than both architecture I & II and over a wider range (0.4-0.95) of offered load. The tendency to saturate for three and four conversations is also less pronounced for architecture III.

Fig. 7(b) shows a comparison of results for nonlocal conversation. For architecture II, the improvement in throughput with offered load over architecture I is less pronounced for the number of conversations that we have modeled. However, note that for four conversations we see an improvement (≈20%) over architecture I in the range of offered loads 0.7-0.9. Thus, the graphs do show a trend in predicting the improvement that is attainable for much larger systems. Unfortunately, given the limitations of existing modeling tools, we were unable to model larger systems. We note once again that architecture III shows a marked performance improvement over the first two architectures. Over the range of offered loads 0.6 and 1.0, architecture III does significantly better than both architectures I and II. The graph suggests that smart bus primitives are as
important for improving the performance of the system for nonlocal conversations as software partitioning.

3) Partitioned Smart Bus: We analyzed a fourth architecture that was motivated by the observation that task control blocks are a shared data structure between the host and the message coprocessor, whereas kernel buffers are a shared data structure between the message coprocessor and the network interfaces. We partition the smart shared memory and the smart bus as follows. The task control blocks are on a partition that interconnects the host with the message coprocessor and the kernel buffers are on a partition that interconnects the message coprocessor with the network interfaces.

We found in all cases that the partitioned organization did not perform significantly better than architecture III. We would have expected such an improvement in performance if there was a considerable contention for the shared memory. These performance results indicate that access to the shared memory is not the bottleneck in limiting the performance. For the same reason, for a given architecture, we do not expect a multiported shared memory to perform better than a single-ported shared memory for any of the four architectures that we analyzed.

F. Summary

In summary, the graphs show the following.

1) Over ranges of offered loads (0.4-1.0 for local and 0.6-1.0 for nonlocal), partitioning the message-based operating system and providing high-level bus primitives result in improvement in performance over a uniprocessor implementation. Thus, there is a range of mixes of computation and communication in which a message coprocessor is appropriate for improving the performance of the system. We observed that the times for typical system services (measured on a Microvax II running Unix) on such as timer, and reading/writing files, range from 0.2 to 6.1 ms. With a local-message communication time of 4.57 ms on Unix, these service times represent an offered load ranging from 0.96 to 0.43; with a nonlocal communication time of 6.8 ms the corresponding offered loads range from 0.97 to 0.53.

2) For one conversation there is a loss in performance due to software partitioning, but the loss is very small. Improvement in performance with the number of conversations is less than linear due to the finite bandwidth of the message coprocessor.

3) Smart bus primitives improve the performance of the system significantly for both local and nonlocal conversations:

4) Software partitioning, and high-level bus transactions (mirroring operating system functions), are a promising approach to solving the message-passing problem in distributed systems. Future research directions include: extending the performance studies to different communication scenarios, exploring the instruction-set architecture of the message coprocessor, implementing in VLSI the hardware assists identified in this research, and extending our studies to the domain of multiprocessors connected by local area networks.

VI. CONCLUDING REMARKS

Local area networking has enhanced the interest of researchers in experimenting with distributed message-based operating systems. Current research [6], [11] and our own measurements of several operating systems show that interprocess communication (message passing) is roughly two orders of magnitude slower than a simple procedure call. Since system services are requested via message passing, the performance of message-based operating systems depends crucially on the rate of message passing. Our goal in this research was to study the problem of interprocess communication in a distributed system, and suggest a system architecture that improves the performance in this environment. In this paper, we suggested a system architecture that was composed of a software aspect and a hardware organization. Using GTPN as a modeling tool, we showed that software partitioning and high-level bus transactions (mirroring operating system functions) are a promising approach to solving the message-passing problem in distributed systems. Future research directions include: extending the performance studies to different communication scenarios, exploring the instruction-set architecture of the message coprocessor, implementing in VLSI the hardware assists identified in this research, and extending our studies to the domain of multiprocessors connected by local area networks.

REFERENCES

[20] U. Ramachandran, "Hardware support for interprocess communica-
Umakishore Ramachandran received the Ph.D. degree in computer science from the University of Wisconsin-Madison in 1986. Since then he has been an Assistant Professor in the School of Information and Computer Science at the Georgia Institute of Technology. His primary research interests are in computer architecture and distributed operating systems. He has participated in the design of several distributed systems: Charlotte at UW-Madison, Jasmin at BellCore, QuickSilver at IBM Almaden Research Center, and Clouds at Georgia Tech. He was also part of the PIPE computer architecture design team at UW-Madison. The research projects he is currently working on include distributed shared memory, memory management structures for object-based systems, issues in the design of large-scale multiprocessors (such as caches and optical interconnects), reconfigurable supercomputer architectures, and architecture support for message-based systems.

Dr. Ramachandran is a member of the Association for Computing Machinery and the IEEE Computer Society.

Marvin Solomon received the B.S. degree in mathematics from the University of Chicago and the M.S. and Ph.D. degrees in computer science from Cornell University, Ithaca, NY, in 1976. He joined the Department of Computer Sciences at the University of Wisconsin-Madison, where he is currently Professor. He was a Visiting Lecturer at Aarhus (Denmark) University during 1975-1976 and Visiting Scientist at IBM Research, San Jose, CA, during 1984-85. His research interests include programming languages, program development environments, operating systems, and computer networks.

Dr. Solomon is a member of the IEEE Computer Society and the Association for Computing Machinery.

Mary K. Vernon (S'82–M'82) was born in Havre de Grace, MD, in 1953. She received the B.S. degree with Departmental Honors in chemistry in 1975, and the M.S. and Ph.D. degrees in computer science in 1979 and 1983, from the University of California, Los Angeles. In August 1983, she joined the Department of Computer Science at the University of Wisconsin-Madison, where she is currently an Associate Professor. Her research interests include techniques for performance modeling of parallel systems, performance Petri nets, and analysis of multiprocessor design issues.

In 1985, Dr. Vernon received the Presidential Young Investigator Award from the National Science Foundation. She is a member of the C.S. Advisory Board of the Computer Measurement Group, and the board of the ACM SIGMETRICS.