TECHNICAL REPORT 93-07

Sharing of Persistent Data Structures for Distributed Cooperative Work on Versioned 64-bit Address Spaces

Shigekazu Inohara, Keitaro Uehara,
Hajime Miyazawa, and Takashi Masuda

Dec 22, 1993

DEPARTMENT OF INFORMATION SCIENCE
FACULTY OF SCIENCE, UNIVERSITY OF TOKYO

7–3–1 HONGO, BUNKYO-KU TOKYO, 113 JAPAN

TITLE

Sharing of Persistent Data Structures for Distributed Cooperative Work on Versioned 64-bit Address Spaces

AUTHORS

Shigekazu Inohara, Keitaro Uehara, Hajime Miyazawa, and Takashi Masuda

KEY WORDS AND PHRASES

versioned 64-bit address spaces, persistent data, distributed cooperative work

ABSTRACT

In implementing a new kind of applications that support cooperative work of users on distributed computers (such as group software development and group CAD), the management of data structures shared among users is the primary importance. Since the applications often share data structures containing pointers on distributed secondary storage, two issues arise for their efficient implementation. (1) To avoid the possibility of time-consuming relocation of data structures at the time of location conflict between two persistent data structures, the location of shared data structures should be globally maintained at the file system level. (2) For immediate propagation of modification on shared data structures in cooperative works, a low-latency cache coherence protocol should be explored. This paper proposes a system that addresses these issues by introducing versioned 64-bit address spaces and a low-latency cache coherence protocol that exploits the file system interface.

REPORT DATE	**WRITTEN LANGUAGE**
Dec 22, 1993	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
13	18

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT

DISTRIBUTION STATEMENT

First issue 40 copies. This technical report is available via anonymous FTP from `ftp.is.s.u-tokyo.ac.jp` (directory `/pub/tech-reports`).

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku Tokyo, 113 Japan

Sharing of Persistent Data Structures for Distributed Cooperative Work on Versioned 64-bit Address Spaces

Shigekazu Inohara, Keitaro Uehara, Hajime Miyazawa, and Takashi Masuda

Department of Information Science, Faculty of Science
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

E-mail: {ino,ktr,tachi,masuda}@is.s.u-tokyo.ac.jp

Abstract

In implementing a new kind of applications that support cooperative work of users on distributed computers (such as group software development and group CAD), the management of data structures shared among users is the primary importance. Since the applications often share data structures containing pointers on distributed secondary storage, two issues arise for their efficient implementation. (1) To avoid the possibility of time-consuming relocation of data structures at the time of location conflict between two persistent data structures, the location of shared data structures should be globally maintained at the file system level. (2) For immediate propagation of modification on shared data structures in cooperative works, a low-latency cache coherence protocol should be explored. This paper proposes a system that addresses these issues by introducing versioned 64-bit address spaces and a low-latency cache coherence protocol that exploits the file system interface. The system has been implemented as a set of user-level servers on DECstations running Mach 3.0 microkernel.

1 Introduction

A new kind of distributed applications that support cooperative work of users on the computer network has gained emerging interest. For example, applications like group software development environment (SDE) and group CAD accelerate communication among the remote people by enabling them to instantly share the knowledge about the status of the on-going project and to change their actions appropriately.

A number of applications for cooperative work including group SDE and group CAD manipulate complex data structures. "Complex data structures" here include common constructs of modern programming languages: recursive type construction with arrays, records (or structs), unions, pointers, and abstract data types. The applications handle the complex data structures in the secondary storage, which can be hierarchically versioned source and object code of new software products in group SDE or design works in group CAD.

Currently, each of these application is forced to build for their own a platform for the sharing of complex data structures in the distributed secondary storage, incurring the cost of developing this kind of applications. File systems of conventional operating systems or conventional database management systems (DBMSs) cannot be direct platforms for these applications for two reasons.

(1) User processes cannot *directly* share data in the file systems or databases. Process's address space and the secondary storage are separated in conventional systems (through UNIX read() and write() system calls or DBMS transactions), fine-grained sharing of data for cooperative work applications is inhibited because of large delay for propagating modification of shared data from one user process to others. The overhead of coping data between processes and the secondary storage is also unavoidable. Another problem of conventional file systems is that in widely-used distributed file systems [17, 11], serializability or the sequential consistency of distributed file caches are not promised. Although weakened consistency models for distributed file caches provides higher performance than the sequential consistency does, this approach makes it difficult for applications with fine-grained sharing of distributed data to rely the facility of data sharing on the underlying distributed file systems.

(2) Pointers in complex data structures in the secondary storage are not supported by the system. Even if plain data in the secondary storage are directly shared among processes through memory-mapped file interfaces [6, 8], pointers cannot be simply mapped from the secondary storage to processes' address spaces. Since a pointer contains an address in the address space of a single process, the pointer is not meaningful in general if it is mapped to other processes. In addition, in most existing systems, the size of a process's address space (typically less than 2^{32} bytes) is not enough for uniquely addressing all the data in the secondary storage. Consequently pointers must be converted between a form in process's address space and another form in the secondary storage whenever they are transferred between the primary and secondary storage. Even with on-demand conversion of pointers known as "pointer swizzling" [18], the pointer conversion poses high time and space costs because it requires run-time overheads that is proportional to the number of pointers in a shared data, and because it cannot coexist with virtual memory optimization like copy-on-write sharing. Consequently, the pointer conversion should be avoided as much as possible.

This paper proposes a solution to these problems of conventional systems in sharing complex data structures in the distributed secondary storage. The proposal has been implementing as a research distributed operating system, which we call Lucas. To solve the first problem, memory-mapping of files to processes is the only interface for accessing secondary storage data in Lucas. Lucas support the memory-mapping over the network, and provides the sequential consistency for distributed cache coherence. Instead of weakening the consistency model, we improved the performance of the cache protocol by exploiting the fact that the caches are accessed through memory-mapped file interfaces. To solve the second problem, the secondary storage has an address space. The secondary storage with addresses, which is called the *resource space* in Lucas, supersedes the conventional file system. A large address space for addressing data in the secondary storage is provided by the latest 64-bit processors [10, 16].

Even with 64-bit address spaces, pointer conversion are sometimes unavoidable. In general, if a complex data structure containing pointers are copied, the original and the copy cause an address conflict, and should be relocated by converting pointers if the original and the copy are to be used in a single address space. To avoid pointer conversion as much as possible, the resource space is *versioned*; each individual complex data structure has a unique address in the resource space, and all the versions of a single complex data structure share the single address. That is, copies of a complex data structure are assigned the same address as their original, thereby avoiding pointer conversion until a process use them at the same time in a single address space. In addition, to efficiently implement the pointer conversion when it is unavoidable, the resource space maintains referencing relationship among complex data structures. Lucas determines which complex data structures should be relocated when an address conflict occur at run-time.

The versioned 64-bit resource space, memory-mapping interface, and the distributed file caches with high-performance sequential consistency are key concepts of the Lucas operating system. User processes in cooperative work applications can effectively share complex data structures in the distributed secondary storage, The versioned resource space avoids all pointer conversion except for those caused by multiple versions of a single data.

The rest of the paper consists of five sections. Section 2 describes the management of the resource space and the support for pointer conversion. Section 3 describes the implementation of Lucas on DECstation 5000 and its performance. Section 4 discusses related work and Section 5 concludes this paper.

2 Management of Complex Data Structures with Regions and Inter-Region References

In Lucas, regions and inter-region references enable users to exploit wide address spaces of 64-bit processors (Figure 1). All the data in the system, including persistent complex data structures, are maintained as *region* in the resource space, and *inter-region references* represent referencing relationship among regions.

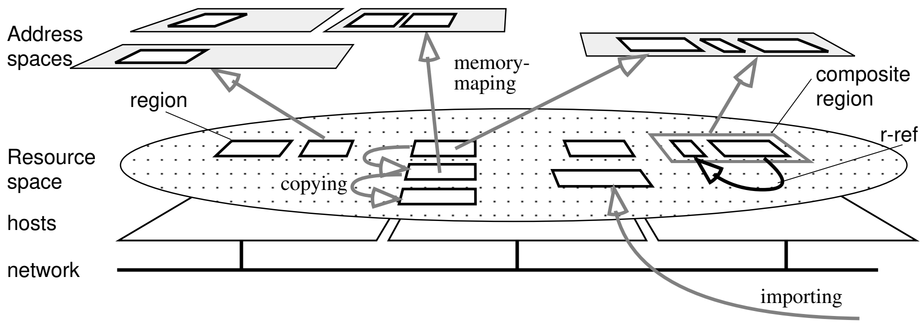

Figure 1: Memory hierarchy of Lucas.

Region A region, consisting of a set of contiguous virtual memory pages, is the principal unit of virtual memory management, secondary storage management, naming, and protection in the Lucas operating system. Regions can be volatile or persistent. Memory-mapping is the only method to access the contents of a region; while a region is mapped to a process's address space, threads in the process can read and write the contents of the region with ordinary memory references. Each independent region occupies a unique address in the resource space, except that the multiple versions of a region are given the same address, and is mapped to the same address of processes' address spaces. As a result, processes can store complex data structures containing pointers into regions, and share them with other processes without pointer conversion.

Inter-region references (or *r-refs* for short) represents the referencing relationship among regions. The existence of incoming r-refs to a region means that the region is referred to be other regions. We introduced r-refs for the support of efficient pointer conversion at the time of relocating regions on an address space.

2.1 Location Management of Regions on the Resource Space

The address range (or simply location) of regions on the resource space are managed by the Lucas system so that the pointer conversion is minimized. At the time of region creation, the location of a new region is not fixed, but *is determined by the system when the region is first mapped to an address space*, as described below.

Users of the Lucas system create a new region by issuing a region_create system call. The region_create call, given the initial size and protection modes as arguments, stores a new region with zero-filled pages in the resource space. In order to give the Lucas system the maximum freedom of determining the location of regions, users do not provides the address in the resource space to the newly created region. The region_duplicate call is another way to create a region. The call makes a new region by copying the contents and outgoing r-refs of an existing region.

We have introduced this interface for versioning regions on the resource space; all the regions created through the region_duplicate call are maintained as the multiple versions of a region and assigned a single address.

Three reasons convinced us to defer the assignment of a unique location in the resource space to a newly created region until its first memory-mapping. First, the run-time overhead of handling regions that do not require unique locations, such as plain text files and private data structures that will never be shared, should be the same as in the conventional systems. Second, the spatial cost of the in-kernel data structure that maintains all the unassigned locations can grow in proportion to the number of location-assigned regions and can be very large because an address space is 2^{64} bytes. Third, finding a globally unique location sometimes involves network communication in a distributed environment, and can take several milliseconds.

To provide a set of interfaces that fully support cooperative work applications without slowing down conventional applications, Lucas offers two modes of mapping. Users control whether the system fixes the location of a region or not using the two mapping modes, 'pinned' and 'unpinned' modes. The pinned mapping mode of region_map maps a region to a globally unique location. A user uses this pinned mode if the region is already location-dependent, or if it can be location-dependent after it is mapped (that is, if he is intended to create pointers to the region after it is mapped).

- When region x is to be mapped and is not assigned a globally unique location, region_map finds a globally unique location and assigns it to x. Otherwise, if x is already location-dependent, region_map uses the assigned location of the region.

- When x is unmapped, region_unmap relinquishes the assigned unique location of x if and only if x is not location-dependent (*i.e.*, x has no incoming pointer-refs).

The unpinned mapping mode of region_map, in contrast, maps a region to a *locally unique location*, that is, the location that does not conflict with any other regions in the target address space. Lucas dedicates a range of each address space for efficient implementation of the unpinned mapping.[1]

- When region x is to be mapped, region_map assigns x a locally unique location.

- When x is unmapped, region_unmap relinquishes the locally unique location assigned to x.

[1]For example, we can reserve first 4096 Gbytes of each address space in a 64-bit address space for holding 1024 unpinned mappings simultaneously.

Users restrict themselves not to create pointers to regions mapped with the unpinned mode.

As a result of introducing the two modes of mapping, the Lucas system guarantees that any combination of regions does not cause an address conflict, thereby avoiding pointer conversion, unless the regions are the multiple versions of a region. The system also guarantees that location-independent regions are handled with the same costs as in conventional file systems. The result of performance measurement with the current implementation assures the latter fact, which we describe in section 3.

The major drawback of having two modes of mapping is that we occasionally search for unique locations more than once for a single region if the pinned mapping mode is repeatedly applied to a location-independent region. Two well-known techniques are used to reduce the overhead of this style of multiple searches. (1) The system allocates a large *cluster* of addresses in a per host basis so that each host can find globally unique locations without communicating with other hosts. (2) Each host retains recently relinquished globally unique locations as *hints*.

2.2 Relocation of Regions

Even with 64-bit address spaces, we cannot avoid all address conflicts between data structures, and hence relocation of them. In the storage organization with the versioned 64-bit resource space, an address conflict occur if (and only if) a process tries to use two versions of a single region simultaneously. The address conflict are resolved only with relocation of a region through a set of pointer conversion, namely, rewriting all the pointers to the region after moving it to a certain location that are used by no other regions.

The relocation of region x consists of two different phases: (1) finding a set of regions that refers to x, and (2) rewriting all the pointers in the set of regions after the new location of x is fixed. Although the second phase can be done only by the application or programming language that knows which part of the contents of the regions are pointers, the first phase should be done by the operating system level because only the operating system can maintain the system-wide information to determine the set of regions whose pointers are to be rewritten. Without the support by the operating system, the relocation processing on the distributed environment is essentially a global copying garbage collection of the whole distributed secondary storage in the system, which is unrealistic.

Lucas support the first phase by introducing r-refs. An r-ref represents the existence of pointer-level dependency between two regions. A region is location-dependent if and only if it has incoming r-refs. For example, if a region x contains a B-tree index to an array of data in region y, as in Figure 2, pointers in the B-tree refer to either x or y. Accordingly, users set up two r-refs, from x to x and from x to y. Each r-ref is implemented as a directed link with

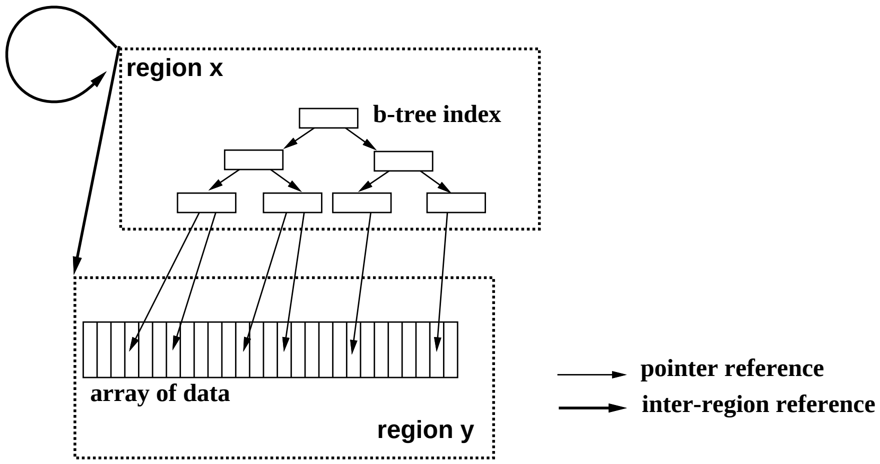

Figure 2: Inter-region references.

a reverse link so that the system can determine at little overhead by which regions a region is referred to.

Since the second phase of the relocation is in general a heavy weight processing, Lucas defers the invocation of the second phase until the regions to be pointer-converted are memory-mapped next time. When region x is to be relocated, the system moves x to a new location, and marks each of the regions that refer to x as "need pointer conversion." If a region with this flag is mapped by some processes later, the system makes an upcall [5] to the predefined user-level routine for pointer conversion, and removes the flag. If a region with the "need pointer conversion" flag has already been used when the relocation of x takes place, all the processes that are using the region is notified of the relocation through another upcall.

3 Implementation and Low-Latency Cache Protocol

The Lucas system has been implementing on DECstations[2] as a user-level server on top of Mach 3.0 microkernels. Fundamental services have already been functional with 47,000 lines of C code.

3.1 Interaction of Region Server, Storage Server, and Cache Server

The resource space, regions, and inter-region references are maintained in cooperation of three servers in each host. A *storage server* maintains the local disks and buffer caches in each host. A *cache server* is a client of local and remote storage servers. Cache servers on the network

[2]DECstations have MIPS R3000 processors currently, and will be upgraded to R4000.

operation	condition	local	remote
map unpinned	CS cache hit	2.1	2.1
map unpinned	CS cache miss	16.7	41.8
unmap unpinned	CS cache hit	1.1	1.1
unmap unpinned	CS cache miss	15.7	40.8
page fault	CS cache hit	2.0	2.0
page fault	CS cache miss	5.9	46.0
page fault	Mach default pager	3.9	—
page access	4Kbyte read	0.4	—

Figure 3: Performance of region-related operations (in msec).

maintains coherent caches of a region by communicating with the storage server holding the secondary storage copy of the region. A *region server* maintains meta-data (i-node information) and inter-region references. A storage server, cache server, and region server is implemented with about 11,600 lines, 15,900 lines, and 19,200 lines of C code, respectively.

The three servers coexist in a single address space for two reasons. First, since communication among the servers are frequent, the overhead of inter-process communication would increase if we implemented them with three separate address spaces. In our implementation, threads of execution frequently cross the boundaries of the three servers for optimizing the interaction among them. Second, the servers share meta-data for regions and inter-region references. By implementing the servers in a single address space, they can share the modification to meta-data without overheads. More significant is the sharing of buffer caches. The storage server and cache server in a single host might, at least conceptually, have separate cache for a single region. The duplication of buffer caches is essentially the duplicated functionality of file systems and virtual memory in conventional operating systems [14]. We solved the duplicated buffer caches by sharing their whole image and integrating the management of buffer caches of the storage server and cache server.

The performance of region-related operations on the current implementation is summarized in Table 3. They are turn-around time of system calls in milliseconds, measured on two DECstation5000s (MIPS R3000 33MHz, 24Mbyte memory, 1.4Gbyte disk) running Mach 3.0 (revision MK82). The map operation with the pinned mode adds 5.3 msec to the corresponding operation with the unpinned mode when the region is first assigned a globally unique location. The additional cost is consumed for registering the unique location to the local disk. The unmap operation also needs the same overhead when the assigned globally unique location is removed. Since we provides the sequential consistency, a remote page accesses in the case of cache miss causes large latency. Local region operations and page operations are optimized as we described above, and their costs are comparative to the costs of similar operations of the default pager, built-in virtual memory manager of Mach.

3.2 Improvement of Cache Performance Using the Region Interface

Since cooperative work applications make fine-grained sharing of distributed data, reducing the latency of remote page accesses has significant effect on the overall performance improvement. Although weakening the consistency model of distributed caches is a well-known approach to enhancing the performance of coherent cache, we did not take this approach because this approach sometimes causes larger delay than the sequential consistency does in propagating updates from one process to other processes. We exploited the two characteristics of regions as memory-mapped files for enhancing the performance of the basic protocol, a read-replica write-invalidation protocol. (1) Each process declares the access mode (either read-only, write-only, or read-write) for accessing a region. (2) Processes declares the session for accessing a region with region_map and region_unmap.

Using these characteristics, we introduced two optimizations: write-mode optimization and read-mode optimization. The write-mode optimization is applied to a process that maps a region with the read-write mode; the cache server regards read accesses from the process as write accesses. With this optimization, the "write after read" style access are handled more efficiently than the basic protocol. The read-mode optimization is applied to a process that maps a region with the read-only mode with a special flag. If the special flag is added to the open mode, the process receives the contents of region resulted from the last session for the region; since the effects of on-going sessions are ignored, the process can access its private copy of the region with 100 % cache hit ratio. At the same time, the performance of the on-going sessions are improved because the number of read-replica are reduced.

In addition, by exploiting the fact that the only way of communication among processes in Lucas is through shared regions, we changed our basic protocol from the strict protocol to the causal consistency protocol [3]. Since processes can observe the difference of the two protocol only if a communication method other than the shared memory is provided by the system, the two protocol is equivalent if processes in an application communicate only within the Lucas system.

We investigated the effects of these optimizations through a couple of simulations that produce various situations in cooperative works. In both simulations, 10 processes in different hosts makes total 100,000 times of accesses to a shared region consisting of 100 virtual memory pages. We changed the ratio of read-only processes and read-write processes from 0:10 (readers ratio 0.0) to 10:0 (readers ratio 1.0). Each read-write process makes a random sequence of read

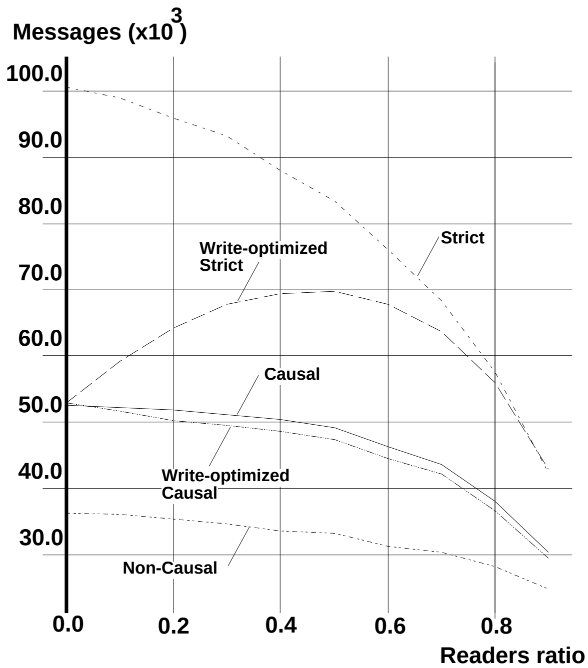

Figure 4: Effect of write-mode optimization.

and write accesses with 60% read and 40% write probability.

The effect of write-mode and read-mode optimization are shown in Figure 4 and 5, respectively. The proposed optimizations are effective in reducing the number of messages during the session, thereby reducing the latency of remote page access. In particular, the causal consistency protocol with read-mode and write-mode optimizations (the new Lucas protocol) reduces 50.7% of messages of the unoptimized strict protocol in average (53.8% in the maximum case). In this case, the improved protocol uses only 1.19 times larger number of messages than the "non-causal" protocol, which is an imaginary protocol that promises the serializability of write accesses but no consistency between read and write accesses.

4 Related Work

Several systems for 64-bit address space computers have already been proposed and are under active prototyping. The Opal system [4] at the University of Washington and the Cubix system [13] at Toshiba provide a flat single address space that is shared by all the processes in the system. Other systems [15, 2] are also under active research. These linear, single address space systems are simple from the user's point of view, and have a large opportunity to gain high performance. However, with the linear single address space, the copying of complex data structures containing pointers *always* requires pointer conversion.

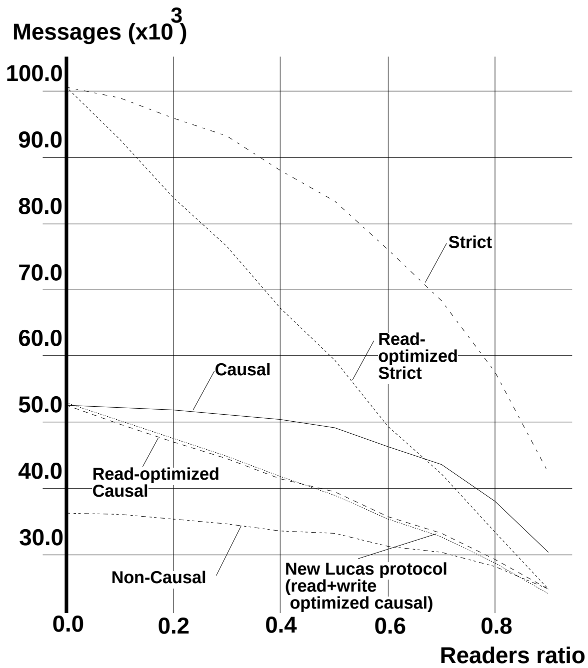

Figure 5: Effect of read-mode optimization.

Eos [7] at INRIA integrates the operating system and persistent programming languages toward efficient management of distributed persistent objects on a shared object space. Lucas, on the other hand, tries to separate programming languages and the operating system by providing a set of language-independent support for region relocation. As a result of the support, various persistent languages and systems, such as PS-algol [1], ObjectStore [9], and Napier [12], will be built with less effort than on conventional operating systems.

5 Conclusions

We proposed an operating system that provides cooperative work applications for a basic framework for sharing complex data structures, which might mutually refers to each other, in the distributed secondary storage. The proposed operating system minimizes the possibility of time-consuming pointer conversion by introducing regions in the versioned 64-bit resource space as the container of complex data structures, supports relocation of complex data structures with inter-region references, and implements high-performance cache coherence protocol by exploiting the interface of regions as mapped files. The proposed system is implemented as a set of servers on Mach 3.0 microkernels, and performance study on the implementation showed that the overall system is implementable with reasonable performance on a common local-area network environment. In particular, the proposed cache coherence protocol is more than twice faster than the conventional sequential consistency protocol.

On the other hand, several issues are yet to be solved. Programming language interface to region and inter-region references, which should be portable to various language platforms, is necessary to realize multi-language persistent programming. External data representation suitable for memory-mapped complex data structures have to be investigated in order to improve storage and run-time performance through conventional techniques such as garbage collection of fine-grained persistent data and hole coalescing of persistent heaps.

References

[1] P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison, "The Persistent Object Managment System," Tech. Rep. PPR-1-83, University of Glasgow Dept of Computer Science, March 1983.

[2] A. Bartoli, S. J. Mullender, and M. van der Valk, "Wide-Address Spaces — Exploring the Design Space," *Operating Systems Review*, vol. 27, pp. 11–17, January 1993.

[3] F. Boyer, "A Causal Distributed Shared Memory Based on External Pagers," in *Proceedings of the USENIX Mach Symposium*, pp. 41–57, Nov 1991.

[4] J. Chase, H. Levy, M. Baker Harvey, and E. Lazowska, "Opal: A Single Address Space System for 64-bit Architecture," in *Proceedings of the Third Workshop on Workstation Operating Systems (WWOS-III)*, IEEE Computer Society, April 1992.

[5] D. D. Clark, "The Structuring of Systems Using Upcalls," Proceedings of the Tenth Symposium on Operating Systems Principles", pp. 128–140, March 1988.

[6] R. C. Daley and J. B. Dennis, "Virtual Memory, Processes and Sharing in MULTICS," *Communications of the ACM*, vol. 11, pp. 306–312, May 1968.

[7] O. Gruber, L. Amsaleg, L. Daynès, and P. Valduriez, "Eos, an Environment for Object-based Systems," in *Proceedings of the 25th Anniversary Hawaii International Conference on Systems Science*, pp. 757–768, IEEE Computer Society, January 1992.

[8] IBM Corporation, "IBM System/38 Technical Developments," Atlanta, CA, July 1980.

[9] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System," *Communications of the ACM*, vol. 34, pp. 34–49, October 1991.

[10] MIPS Computer Systems, Inc., *MIPS R4000 Microprocessor User's Manual.* 950 DeGuigne Drive, Sunnyvale, CA, 94088, 1991.

[11] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: A Distributed Personal Computing Environment," *Communication of the ACM*, vol. 29, pp. 184–201, March 1986.

[12] R. Morrison, A. L. Brown, R. C. H. Conner, and A. Dearle, "Napier88 Reference Manual," Tech. Rep. PPRR-77-89, University of Glasgow and St. Andrews, 1989.

[13] T. Okamoto, H. Segawa, S. H. Shin, H. Nozue, K. Maeda, and M. Saito, "A Micro Kernel Architecture for Next Generation Processors," in *Proceedings of the Micro-Kernels and Other Kernel Architectures Workshop*, pp. 83–94, USENIX Association, April 1992.

[14] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The Sprite Network Operating System," IEEE Computer, vol. 21, no. 2, pp. 23–36, February 1988.

[15] M. Scott and W. Garrett, "Shared Memory Ought to be Commonplace," in *Proceedings of the Third Workshop on Workstation Operating Systems (WWOS-III)*, IEEE Computer Society, April 1992.

[16] R. L. Sites, "Alpha AXP Architecture," *Communications of the ACM*, vol. 36, pp. 33–44, February 1993.

[17] I. Sun Microsystems, "NFS: Network File System Protocol Specification," RFC 1094, Network Information Center, SRI International, March 1989.

[18] P. R. Wilson, "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware," *ACM SIGARCH Computer Architecture News*, vol. 19, pp. 6–13, June 1991.