Operating System Support for High Performance Parallel I/O Systems

Brian Bershad, University of Washington
David Black, Open Software Foundation Research Institute
David DeWitt, University of Wisconsin
Garth Gibson, Carnegie Mellon University
Kai Li, Princeton University
Larry Peterson, University of Arizona
Marc Snir, IBM Watson Research

Abstract

This document describes the operating system support component in the Scalable I/O Initiative. Our efforts cover three critical areas of scalable, parallel I/O for for high-performance multicomputers: networking, memory management, and file and object store. We are leveraging off on-going research results in these three areas. Our main efforts in the scalable I/O initiative will be directed towards hardening, porting, tuning, and deploying our technology on the target platforms.

1 Introduction

This document describes the operating system component in the Scalable I/O Initiative for high-performance multicomputers. The operating system component offers coverage over three critical requirements for message-based massively parallel multicomputers:

Networking : Support low-overhead, high-bandwidth, and scalable communication between processors. This will involve work at two different levels. The first, *intranetworking*, enables communication between processors that are part of the same multicomputer. The second, *internetworking*, enables communication between remote processors connected by networks such as Ethernet, HIPPI, FDDI and ATM.

Memory Management : Provide flexible memory management, including a large shared virtual memory that enables the use of a shared-memory programming paradigm even on message passing machines, memory servers that extend the memory hierarchy of multicomputers by introducing a remote memory server layer between the local physical memory and fast stable storage such as disks, and a checkpointing facility that ensures that long-running programs can be restarted and/or migrated in the case of processor or system failure.

File and Object Store : Provide scalable I/O from tera- and peta-byte secondary and tertiary storage. This component will be comprised of three major tasks. The first involves defining and implementing a standard applications interface to both the Intel Parallel File System (PFS) and the IBM Vesta Parallel System. As an alternative, we will explore whether a parallel, persistent object store that provides transparent access to persistent objects (e.g. large matrices) on secondary and/or tertiary storage can provide satisfactory performance to HPF applications. The third component will be to design and evaluate alternative strategies for prefetching files and objects from tertiary storage into secondary storage.

We have already developed technology that adequately demonstrates the feasibility of our approaches, which are described in the remainder of this section. For this proposal, our main efforts will be directed towards hardening, porting, tuning, and deploying our technology on the target platforms.

2 Related Efforts

The proposed work will all be done in the context of the following systems software, which have been developed under other ARPA contracts:

- The Mach microkernel from Carnegie Mellon University. Mach supports a small number of orthogonal abstractions such as threads, memory, and IPC, on top of which high-level operating system services such as filing, networking, and process management can be implemented. Recently, work at CMU, and now at the University of Washington, has focussed on improving the performance of networking protocols within the context of the Mach operating system[15, 14, 1].
- OSF/1 AD from the Open Software Foundation[18]. AD extends Mach to provide single-node semantics for a distributed memory multicomputer such as the Intel Paragon.
- The *x*-kernel from the University of Arizona [5, 16]. The *x*-kernel provides a platform-independent framework for implementing network protocols. It has already been integrated in Mach.
- The shared virtual memory [10, 11], memory server [6], and checkpointing tools [12, 13] developed at Princeton University. They currently run on iPSC/860 and target to run on the Intel Paragon.
- The SHORE persistent object manager being developed at the University of Wisconsin [2, 3]. SHORE provides a language-neutral, scalable persistent object store target to run on a variety of hardware platforms from networks of workstations to large parallel processors such as the Intel Paragon and the IBM SP2.

Our work will also leverage two particular industrial efforts: Intel's Parallel File System (PFS) and IBM's Vesta Parallel File System. In addition, we will be defining interfaces that will allow us to access the mass storage systems being designed by the National Storage Laboratory (NSL). NSL is a joint industry, DOE national laboratory, university collaboration with over 20 participants organized to investigate, demonstrate, and commercialize high performance hardware and software to remove bottlenecks in the I/O performance and functionality of very large, hierarchical mass storage systems.

3 Networking

We propose to design, implement, and evaluate techniques for providing high-performance networking support of multicomputers. Our effort will involve both communication among the nodes within a distributed memory multiprocessor (*intranetworking*), and between a particular multiprocessor and remote machines via a high-speed network such as FDDI, ATM, or HIPPI (*internetworking*). Much of the work described in this section will be done in the context of the x-kernel protocol implementation framework.

3.1 Protocol Framework

We have implemented a protocol framework, called the x-kernel [5, 16], that supports the rapid implementation of efficient network protocols. We have integrated the x-kernel protocol framework into the Mach 3.0 operating system, in a way that allows a protocol graph to run across multiple protection domains, including the Mach microkernel, a network server, and application domains. Various protocol suites, including TCP/IP and Mach IPC have been implemented in the x-kernel [17].

The x-kernel provides protocol implementors with an interface that completely hides the details of the rest of the system. That is, communications software is decoupled from the application programming interface, the host operating system, the underlying architecture, the details of the network adapter board, the protection/security domain that it runs in, and the processor it runs on. As the result of this design, the networking subsystem is highly configurable. System administrators configure exactly the suite of network protocols that are needed, and then place those protocols in different protection domains and on different processors depending on how they are willing to trade performance against trust.

We achieve good performance through the vertical integration of the entire communications subsystem, from the application interface, through the protocol stack, to the network adapter board. This implies that all performance critical decisions are made in a coherent fashion, thereby avoiding the expense of changing abstractions as network packets flow through the system. In other words, by implementing all protocols in the context of the x-kernel, we are able to identify and optimize the critical path.

The x-kernel framework has been demonstrated on workstations connected to high-speed networks (FDDI and ATM) [4]. We propose to port the x-kernel to OSF/1 AD running on the Intel Paragon, and to tune/optimize its performance in support of the communications protocols described throughout this section.

3.2 Intranetworking

Intranetworking is concerned with moving data between processors that are connected to the same logical system. Intranetworking may be within or between parallel programs. For communication within a parallel program, latency and throughput are the primary metrics of success. For communication between programs, such as occurs during client/server operating system interaction, protection and authentication become as important as performance. As two styles of communication must be accommodated, we propose using two distinct solutions.

Unprotected Communication

For communication within a logical process, we will adapt a set of low-overhead networking protocols that we have developed for local-area ATM (Asynchronous Transfer Mode) networks [1]. These protocols have the following important characteristics: they require minimal hardware support from the networking interface, incoming/outgoing data is touched by the CPU only once as it enters/leaves the network interface, they have extremely modest CPU requirements during the common case of reliable/sequenced delivery, and they support flow-control and congestion-control with low overhead.

Our protocol architecture yields high performance by eliminating protocol redundancy and by exploiting common-case communication behavior. With this approach, we can combine the functionality typically found in four separate layers of the ISO model – data link through session – in a single pass over the data, delivering high throughput and low latency. Our protocol architecture requires minimal hardware support from the network interface and switch fabric, yet efficiently provides services such as segmentation and reassembly, flow control, congestion avoidance, and error recovery. We have implemented our protocol architecture on a switch-based ATM network consisting of DECstation 5000/200 workstations running the Mach 3.0 operating system. Our implementation achieves latencies and bandwidths close to the physical limitations imposed by the hardware, yet offers applications a high-level reliable transport interface [1]. Currently, these protocols run over a switch-based ATM network. The task of adapting them to the Paragon mesh would be a part of this contract.

Protected Communication

For protected communication, such as that employed by client-server interactions, we propose utilizing technology from the x-kernel's implementation of Mach IPC on loosely-coupled networks [17] to improve NORMA IPC for the Paragon and other platforms. Mach IPC is the communication abstraction of the Mach operating system. It supports a rich semantics: multi-part, typed messages are delivered reliably and in sequence to ports; tasks hold, and can transfer to each other, the right to send to, and receive from, ports; and tasks are notified when the status of ports they hold a right to changes (e.g., the holder of a send right is notified when no task currently holds a receive right). NORMA IPC is the transparent kernel to kernel extension of Mach IPC used on Paragon (among other machines). The $x$kernel's protocols for port right management in the presence of possible node failure allow NORMA IPC to respond to node failures (e.g., by delivering notifications that result from port rights on that node being destroyed). This improves NORMA's current behavior (no resilience to node failures) and allows higher levels of system software to rely on IPC notifications as part of their recovery mechanisms. Components of the *x*kernel's protocol implementations may also be useful in porting NORMA IPC to additional hardware platforms such as the SP1.

3.3 Internetworking

This component provides scalable, flexible internetworking (e.g., TCP/IP) technology, where networking performance is limited only by the path from the application to the network interface hardware. Operating system interference is kept to a minimum because we eliminate the operating system on critical send and receive operations.

The key insight behind our work is that an application's interface to the network is distinct and separable from its interface to the operating system. By separating the interfaces, we can provide a fast path between the application and the raw network media while maintaining the semantics of operating system abstractions specified by standard application programming interfaces. Specifically, code in the application address space implements the network protocols and transfers data to and from the network, while an operating system server manages the heavyweight abstractions that applications use when manipulating the network through operations other than send and receive. We achieve high performance by avoiding protection boundary crossings, data copying, and unnecessary software layers in the important common case of send and receive. We provide flexibility because the user-level networking software may be developed, configured, and specialized independently from the rest of the operating system.

Our approach is scalable to thousands of processors because each processor can maintain its own copy of the network protocol stack and protocol endpoint data structures. Thus, logically distinct protocol endpoints are also structurally distinct, and can therefore run in parallel.

Our approach of separating the protocol implementation into two pieces, one fast that resides in the application's address space and provides network connectivity, and one complete that resides in an operating system server and that provides full interface compatibility, has resulted in substantial performance improvements relative to a server-based implementation for uniprocessor implementations based on the Mach operating system. More importantly, our user-level protocol libraries achieve performance (both throughput and latency) that is comparable, and in some cases superior, to well-tuned kernel-based implementations. Furthermore, this approach is well suited for loosely-coupled environments since each application has a private copy of the network protocol stack which means network data transfer requires no interaction with an operating be system server. Thus, networking performance is limited only by the path between the application and the network interface hardware.

The internetworking protocols themselves will be implemented using the *x*-kernel. Currently, our distributed protocol implementation relies on the Berkeley UNIX protocol suite, although this is incidental and not fundamental to our mechanisms which are primarily concerned with interfaces between the protocols and the operating system, as opposed to within the protocols themselves. Consequently, we do not expect that retargeting our implementation to the *x*-

kernel will be a significant problem.

4 Memory Management

The goal of this component is to incorporate new techniques for memory management that are necessary for the current and next generations of massively parallel multicomputers. The techniques include shared virtual memory, remote memory servers, and checkpointing.

4.1 Shared Virtual Memory

Shared virtual memory [10, 11] implements coherent shared memory on a multicomputer without physically shared memory. The shared virtual memory system presents all processors with a large coherent shared memory address space. Any processor can access any memory location at any time. The shared memory address space can be as large as the memory address space provided by the MMU of the processor. The address space is coherent at all times, that is, the value returned by a read operation is always the same as the value written by the most recent write operation to the same address. In conjunction with the memory servers, the shared virtual memory provides users with large shared virtual memory spaces.

4.2 Memory Servers

Memory servers [6] for multicomputers allow sequential programs, message-passing programs, and shared virtual memory programs to use the entire physical memory resources effectively. The memory server model extends the memory hierarchy of multicomputers by introducing a remote memory layer whose latency lies somewhere between local memory and disk. A memory server is a multicomputer node whose memory is used for fast backing storage and logically lies between the local physical memory and fast stable storage such as disks. The memory server model takes advantage of both fast routing networks and available memory resources in multicomputers. A page transfer on the state-of-the-art multicomputer is three orders of magnitude faster than a page transfer between memory and disk. The performance advantage of the memory server mechanism over the traditional virtual memory management will become even more significant as the performance gap between routing networks and secondary storage widens.

4.3 Checkpointing

The goal of this component is to support low-overhead (space and time) checkpointing and restarting for programs running on a massively parallel multicomputer. Checkpointing allows long running jobs to save their state from time to time so that they can be restarted in case of failures, or in case of job swapping due to resource allocation. A checkpointing mechanism must be both space and time efficient. Existing checkpointing systems for MPPs checkpoint the entire memory state of a program. The space requirements for such an approach is proportional to twice the size of the data structures in a program (the previous checkpoint must be preserved during a checkpoint operation). Similarly, existing checkpointing systems work by halting the entire application during the construction of the checkpoint. This stoppage can have a substantial negative impact on the total execution time.

Our research in the area of checkpointing has resulted in a set of algorithms for generating compact, low-latency checkpoints[12]. Checkpoints are compact because only the data changed since the previous checkpoint must be written to stable store (as opposed to the entire image). Checkpoints have low-latency because they are generated *concurrently* during the program's execution.

Our techniques are applicable to multicomputers that do not have explicit hardware support for message synchronization. Distinct memory address spaces on MPPs make it difficult to stop a program in a consistent state, because there may be asynchronous messages still in transit through the network while the snapshot is being taken. The synchronization methods developed at Princeton and the University of Wisconsin [13] require a minimal number of synchronization messages and loggings for MMPs based on static wormhole routing networks.

5 File and Object Stores

This component is comprised of three major tasks including including defining and implementing a standard applications interface to both the Intel Parallel File System (PFS) and the IBM Vesta Parallel System, exploring the viability of a parallel, persistent object store that provides transparent access to persistent objects on secondary and/or tertiary storage, and techniques for prefetching files and objects from tertiary storage into secondary storage.

5.1 Parallel File Systems Support

Our efforts in the parallel file system area will be consist of four major thrusts:

- Develop a comprehensive benchmark suite for parallel file systems.
- Design a common interface for the Intel PFS and IBM Vesta parallel file systems.
- Integrate resulting secondary storage file system with standard tertiary storage systems
- Enhance performance of OSF1/AD file system.

Parallel File System Benchmark Suite

Together with the project component responsible for characterizing I/O characteristics of parallel applications, we will on developing a comprehensive benchmark suite for parallel file systems.

While a limited evaluation of the Intel Parallel File System (PFS) has been conducted[8], no common, comprehensive benchmark for parallel file system performance exists. We propose to develop such a benchmark. Experience in the database system area has proven that such benchmarks are invaluable to obtaining the maximum performance out of a system.

Such a benchmark will serve several functions. First it will be used to understand the performance characteristics of the current Intel Parallel File System (PFS) and the IBM Vesta parallel file system. The results we obtain from speedup and scalability testing of both file systems will be used to locate performance bottlenecks, to explore the effectiveness of alternative design choices, and to guide future development of both file systems.

A Common Interface for the Vesta and PFS File Systems

One of the major reasons that Unix has been such a commercial success is that all implementations supported a standard file system interface. This is not true in the area of parallel file systems. For example, Intel's PFS tries to preserve the Unix-file system interface. Five different access modes are provided[7], ranging in complexity from having each compute node maintain its own file pointer to a very centralized mode in which a single compute node reads the file and then distributes what it reads to the other compute nodes. Three of the five I/O modes requires a centralized coordinator. This coordinator is likely to become a bottleneck as the system is scaled. Files in PFS may be striped across multiple storage units (a single disk or a RAID device) but all files in the same file system must span the same storage units and must have the same striping factor (the number of logically contiguous blocks stored on a single storage unit before proceeding to the next storage unit). Files can only be striped in a round-robin fashion.

The functionality provided by Vesta is significantly different from PFS. First, Vesta does not attempt to preserve the standard Unix file system interface. Files are created via an explicit create call instead of by overloading the write system call. When a file is created, the physical layout of the file is specified. The layout information includes the starting storage unit, the number of storage units to use, and the striping factor for the file. Each file in a file system may be striped differently. No centralized catalog is used to keep track of the layout information. Instead, the file name is hashed to storage unit to locate the meta file containing the layout information for the file.

A second unique feature of Vesta is that when a file is opened, the application can specify what logical partitioning is desired. Basically the logical partitioning provides a view for the application to use when accessing the file. By setting the parameters to the open call appropriately, a variety of useful views are possible including row, column, block and block cyclic decompositions of two dimensional matrices. The application operates on the file through its logical view of the file. The underlying software takes care of mapping the logical view to the physical layout of the file - all without the use of a centralized coordinator. Another unique feature of Vesta is that files can be checkpointed, enabling updates to be rolled back in case an application fails.

In collaboration with the language/compiler community, we will develop a common interface to Intel's PFS and the IBM's Vesta. A common interface is critical to achieving portability of applications between the two systems. We will begin by studying and benchmarking both systems as well as examining the needs of various file system clients such as extensions to HPF to support "out-of-core" arrays. It is too early to state exactly what this common interface will look like. One alternative might be to implement a PFS-compatible interface on top of Vesta. On the other hand, since Vesta has significantly more functionality than PFS, implementing a Vesta-compatible file system (including checkpointing) for the Paragon might make more sense.

Support for Tertiary Storage

We also intend to investigate strategies to integrate local parallel file systems and persistent object stores with large scale tertiary storage. Our approach will be to work closely with the National Storage Laboratory's (NSL's) High Performance Storage System (HPSS) project to assure the appropriate mass storage interface and other functionality to meet the Scalable I/O project's application and other prototype system requirements. HPSS will be integrated into the two Scalable I/O testbeds at Argonne and Cal Tech beginning in 1995.

OSF1/AD File System Enhancements

We intend to also develop a flexible framework to support the implementation of high performance parallel file systems. This should increase the ease of experimentation and interchange of results. The flexibility in this framework will include:

- Caching policy. We will extend the AD file system to support read-ahead, write-behind, and eviction. We will provide interfaces to accept caching information from the application, including the language runtime system.
- Server residency. We will investigate alternatives for providing high performance file access to applications while minimizing the file system code that must reside on nodes performing computations.
- Decomposition. The framework will support the decomposition of a file system into modular components, enhancing configurability, and encouraging reuse. Among the file system components amenable to modularization are single system image support for a multicomputer, and caching based on techniques described in the Memory Management section.

Our pahis area has produced techniques for the coexistence of multiple access methodologies (direct access fast path, mapped files, buffer cache) and a framework for supporting a parallel file system independent of the underlying (disk managing) file system. We intend to incorporate this functionality into the proposed framework. In designing this framework, past work on stackable vnode architectures will be carefully examined and used as a basis for the framework if it can provide a reasonable match to supercomputer I/O requirements.

In the area of infrastructure, we expect to implement support for extent-like behavior on large transfers for the existing UFS-based implementation of AD's file system. This support will be based on techniques such as larger block sizes (including the use of fragments for meta-data), and coalescing of blocks to avoid performing multiple device operations for I/O requests that involve multiple file system blocks. Such suppted optimizations provide further improvements to the performance of the fast path for the large transfers expected in a supercomputer.

5.2 A Parallel, Persistent Object Store Alterative

As an alternative to a conventional parallel file system, we also propose to pursue a much more radical approach. It is our contention that scientific applications should NOT deal with the file system in terms of physical blocks of data (e.g. read the next 512 bytes). As an example, consider a very large 2D array. Today the programmer is responsible for mapping pieces of the array to the file system. In the case of Intel's PFS this process is tedious, error-prone, and very inflexible. While Vesta's higher-level interface makes this mapping much simpler, we feel that applications should deal with mass storage at even a higher semantic level. In particular, we propose to replace the conventional file system with a parallel persistent object store such as the one being developed as part of the ARPA-funded SHORE project[2]. There are a number of reasons why we advocate such an approach.

First,the languages for such systems allow programmers to manipulate both transient and persistent data (e.g. a matrix or a complex data structure) in the same way, freeing the programmer from having to do explicit file I/O. The CAD/CAM community has already begun to realize significant gains in programmer productivity through the use of such persistent programming languages. This community has also demanding performance requirements and they have found that this increase in productivity does not imply a corresponding loss in performance. The High Performance Fortran language has features that should mesh well with this approach.

Second, too much important information is lost when data is stored Unix-like byte-stream files. A scalable I/O system based on the concept of typed persistent objects will provide a higher level semantics for the data as the application programmer will be provided a full type system for describing his/her persistent data. In addition, since the type descriptor for each persistent object is itself a persistent object, the meta information about the actual data will never be accidentally displaced.

Third, very large objects can be transparently partitioned across multiple disk drives. As an example, assume the existence of a 2D persistent matrix class. Associated with the class will be a set of predefined methods for storing and manipulating instances of the class. As an example consider the create method (i.e. new() in C++). As parameters, this method might accept both the number of disks used to store the object and the declustering strategy for mapping the elements of the matrix to mass storage. For example, a "row-wise" declarative would distributed the elements of each row across the specified number of drives.

In such an approach, the actual computation (e.g., an eigensolver) would itself be a method accessing the matrix as if it were memory resident (regardless of its size). By suitably redefining the subscripting operator (i.e. [ ]) to incorporate the declustering factor and strategy, access to elements of the matrix can occur transparently, regardless of what disk actually holds the desired data. Such an approach avoids having the declustering factor or strategy from being "wired" into the application, allowing the user to tune the application by modifying each without having to rewrite the application itself (only the data will have to be reorganized). We plan on developing a full set of such classes for use in numerical applications.

We will use software being developed as part of the SHORE (Scalable Heterogeneous Object REpository) project[2] as the basis for this component of the project. The goal of the SHORE is to develop a persistent object system capable of satisfying both object-oriented database system applications (e.g. CAD) as well as traditional legacy applications (e.g. compilers, editors, ...) that currently depend on the Unix file system. SHORE uses a symmetric, peer-to-peer distributed architecture which eliminates the distinction between client and server processes. In SHORE, every participating processor (whether or not the processor has database volumes attached) runs a SHORE server. This makes the software scalable. It can run on a single processor, a network of workstations, or a large parallel processor such as the Intel Paragon or IBM SP2.

SHORE's type system is based on ODL, a language-neutral, object definition language for persistent object systems that has been defined by the vendor consortium ODMG [3]. Typing persistent objects simplifies the task of supporting heterogeneous hardware environments and makes it feasible to support access to persistent objects from multiple programming languages. Currently support for C++ and Ada are planned. As part of this effort, we design and implement a language binding for HPF. Furthermore, in conjunction with the language/compiler team, we will extend the current SHORE mechanisms so that out-of-core arrays in HPF can be efficiently supported by SHORE. In order to make the integration with HPF as transparent as possible, we will study the use of shared-virtual memory as an implementation strategy. The use of virtual memory faulting techniques has proven very successful in the implementation of object-oriented database systems[9]. These techniques make accessing persistent data transparent to the programmer. Using a combination of memory faulting techniques and the shared-virtual memory mechanisms discussed above should enable us to provide transparent access to "out-of-core" HPF arrays that have been striped across multiple storage units.

6 Interactions With Applications

We will have close interactions with the application researchers in designing application interfaces and developing parallel I/O benchmark suites. Such interactions will be iterated throughout the Scalable I/O initiative effort.

Our first step is to work with application scientists in the Scalable I/O initiative to compose a suite of parallel I/O benchmarks using traditional file system interfaces. Although using the traditional Unix file system interface may not be the ultimate interface for the parallel I/O requirements in the Grand Challenge applications, these parallel I/O applications can serve as an initial quantitative measurement benchmark suite. We can use it to measure and improve our interface designs and enhance the implementation of our systems effort in networking, memory hierarchy management, shared virtual memory, and persistent object store.

As the interaction among the operating system group, application group, language group and performance measurement group proceed, we will iteratively develop the parallel I/O benchmark suites based on newly developed and improved interfaces to networks, memory hierarchy management, file systems, and persistent object storage. We will use the newly developed benchmark suites to validate and test our systems on the hardware platforms at scale.

7 Plan and Schedule

We will be porting most of our systems software as soon as the project starts:

- Debugging support for Paragon, including CMU's TTD, from OSF after 3 months.
- Port of x-kernel to OSF/1 AD from Arizona after 9 months.
- Port of user-level TCP/IP to x-kernel running in Unix Server from Washington after 9 months.
- Port of memory servers to Intel Paragon, 12 months from start.
- Initial version of a checkpointing tool that allows users to specify checkpointed data completes 12 months from start.
- Port of shared virtual memory to Intel Paragon, 24 months from start.
- An initial version of OSF/1 AD that includes the x-kernel, user-level TCP/IP, and an initial set of file system enhancements will be released by OSF after 1 year. Subsequent releases every 12 months.
- Port of SHORE persistent object repository to Intel Paragon, 6 months from start.
- Develop a comprehensive benchmark suite for parallel file systems, 9 months from start.
- Design a common interface for the Intel PFS and IBM Vesta parallel file systems, 12 months from start.
- Enhance performance of OSF1/AD file system, 18 months from start.

After initial porting effort of various systems software, we will work together with application, lanaugage, performance measurement groups to iteratively derive interfaces and enhance systems performance for applications.

References

[1] Jose Brustoloni and Brian N. Bershad. Protocol Processing for High-bandwidth Low-latency Networks. Available as a CMU-CS Technical report CMU-CS-93-132. April 1993.

[2] Michael Carey et. al. Shoring Up Persistent Applications. Submitted to the 1994 ACM SIGMOD Conference, December 1993.

[3] R. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Mateo, CA, 1993.

[4] Peter Druschel and Larry L. Peterson, Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages189–202, December 1993.

[5] Norman C. Hutchinson and Larry L. Peterson, The *x*-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[6] Liviu Iftode, Kai Li and Karin Petersen. Memory Servers for Multicomputers. In *IEEE COMPCON Spring '93*, pages 538–547, February 1993.

[7] Intel Supercomputer Systems Division, Using Parallel File I/O. In Chapter 5, *Paragon User Guide*, November 1993.

[8] , Intel Supercomputer Systems Division, Parallel File System Performance. In Chapter 4, *Paragon System Software Release 1.1 Release Notes for the Paragon XP/S System*, November 1993.

[9] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10), October 1991.

[10] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transaction on Computer Systems*, 7(4):321–359, November 1989.

[11] Kai Li and Richard Schaefer. A Hypercube Shared Virtual Memory System. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume I, pages 125–132, August 1989.

[12] Kai Li, Jeffrey F. Naughton, and James S. Plank. A Real-Time, Concurrent Checkpoint and Recovery Algorithm For Parallel Programs. In *Proceedings of the 1990 ACM Symposium on Principles and Practice of Parallel Programming*, Seattle, Washington, March 1990.

[13] Kai Li, Jeffrey F. Naughton, and James S. Plank. An Efficient Checkpointing Method for Multicomputers with Wormhole Routing. In *International Journal of Parallel Programming*, 20(3):159–180, June 1991.

[14] Chris Maeda and Brian N. Bershad. Networking Performance for Microkernels. In *Proceedings of the Third Workshop on Workstation Operating Systems*, Pages 154–159, April 1992.

[15] Chris Maeda and Brian N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the 14th ACM Symposium on Operating System Principles.*, Pages 244–255, December 1993.

[16] Sean W. O'Malley and Larry L. Peterson. A Dynamic Network Architecture. In *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[17] Hilarie Orman and Ed Menze and Sean O'Malley and Larry Peterson, A Fast and General Implementation of Mach IPC on a Network. In *Proceedings of the Mach Usenix Symposium*, February 1993.

[18] Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, and Durriya Netterwala. An OSF/1 Unix for Massively Parallel Multicomputers. In *Proceedings of Winter 1993 Usenix Technical Conference*, San Diego, Pages 449-468, January 1993.