ISTORE: Introspective Storage for Data-Intensive Network Services

Aaron Brown, David Oppenheimer, Kimberly Keeton, Randi Thomas,
John Kubiatowicz, and David A. Patterson
Computer Science Division
University of California at Berkeley
387 Soda Hall #1776
Berkeley, CA 94720-1776
{abrown,davidopp,kkeeton,randit,kubitron,patterson}@cs.berkeley.edu
http://iram.cs.berkeley.edu/istore/

Abstract

Today's fast-growing data-intensive network services place heavy demands on the backend servers that support them. This paper introduces ISTORE, an intelligent server architecture that couples LEGO-like plug-and-play hardware with software adaptability and continuous monitoring. ISTORE exploits introspection *to provide high availability and performance while drastically reducing the cost and complexity of administration. ISTORE provides a generic server platform designed to be specialized to match the exact demands of a specific network service application, providing the benefits of a fully-custom-built server, or appliance, without the complexity of constructing one from scratch. Additionally, once specialized, ISTORE acts dynamically to monitor and adapt to changes in the imposed workload and to unexpected system events such as hardware failure. This adaptability is enabled by a combination of intelligent self-monitoring hardware components and a software extensibility mechanism that allows the target application to specify adaptation policies to the system using constrained, domain-specific languages.*

1 Introduction

We are entering a new era of computing, one in which traditional PCs, workstations, and servers are giving way to "thin" clients backed by powerful, distributed, networked information services. Information has become *the* commodity, with total growth in online data storage more than doubling every nine months [19]. This era of data-intensive network services is being driven by new classes of applications such as electronic commerce, information search and retrieval, and online decision support. Such online services impose stringent requirements that are difficult to achieve using today's commodity computing technologies: they require availability that permits a mean time to failure measured in no less than years, performance that allows for simultaneously servicing of millions of clients, and scalability that meets the rapid growth of the user base and the data those users need to access. Moreover, studies show that, especially in this demanding environment, maintenance and administration costs are enormous, consuming anywhere from two to twelve times the cost of the hardware [10][11][12][26]. Such maintenance costs are inconsistent with the entrepreneurial nature of the Internet, in which widely utilized, large-scale network services may be operated by single individuals or small companies.

In this paper we turn our attention to the providers of new data-intensive services, be they Internet-based organizations or data centers used internally by a company. We argue that continuous monitoring and adaptation, or *introspection*, is a powerful technique that can be exploited to build high-performance data appliances that administer themselves. Introspection allows the system to automatically achieve a number of goals. At the lowest level, the system can examine data access patterns and media integrity, responding to hot-spots or device failures when they occur by migrating and replicating data. At a higher level, the system can make predictions about future access patterns and can use these predictions to guide global rearrangement of data layout to optimize future performance, to dynamically trade off access latency and storage efficiency, or to maximize data integrity in the face of impending media failure.

This paper presents the architecture of the ISTORE intelligent storage *meta-appliance*, a system that places introspection and its associated adaptability as its architectural cornerstones. By a meta-appliance, we mean a storage infrastructure composed of hardware and software that can be flexibly customized to create an information appliance uniquely adapted to its application niche. ISTORE-based data appliances are designed to provide exactly one application or service and, as a consequence, enjoy full cooperation among all levels of the system, from the hardware to the runtime system to the application. These components work in concert to provide application-appropriate semantics, optimizations, and responses to external stimuli such as component failures. Although different appliances share similar base mechanisms, what differentiates them is the application-specific policies that define when various mechanisms are invoked, what semantics they guarantee, and on what data they operate.

While a variety of data storage appliances have been developed during the past decade [13][24], the downside of existing methods for constructing appliances is that each new application or service requires the construction of a complete, integrated system, usually from general-purpose commodity hardware and software components. ISTORE's meta-appliance approach avoids the general-purpose system's lack of focus by allowing customization of a simple but powerful system that provides as its base the runtime mechanisms needed to implement a complete appliance. The interfaces through which customization of the ISTORE system occurs are a unique combination of domain-specific languages and downloaded user code. These interfaces achieve many of the advantages of fully general, user-level operating system customizations [2][14][22] while avoiding much of the complexity usually associated with general extensibility. In ISTORE, the range of

possible customizations stretches from the details of scheduling and layout policies to the appropriate response to component failure.

The base ISTORE system consists of intelligent LEGO-like plug-and-play hardware components (such as disks, blocks of memory, and compute nodes) that can be hot-swapped into a scalable, fully redundant hardware backplane. Each of the hardware blocks in an ISTORE system includes an embedded processor that is responsible for observing and reacting to the unique runtime behavior of its attached device. As such, these processors comprise essential components of ISTORE's introspective substrate. Application-specific code can also be downloaded into these processors to provide scalable data access primitives such as database scan, sort, and join.

Hardware faults and failures are handled automatically by ISTORE's introspective runtime system. Faults are entirely self-healing, as the system automatically shifts load from non-functional components to working components. The exact policies that are invoked during failures can be customized for a particular application. Furthermore, data layout and replication are under the complete control of the introspective facilities. As a result, human maintenance of an ISTORE system consists of nothing more than adding new hardware when resources become low (a situation detected automatically by the introspective runtime system) or replacing faulty hardware within a reasonable amount of time after failures occur—all other aspects of administration, including performance tuning, are handled automatically.

ISTORE's introspective nature sets it apart from today's existing general-purpose server architectures. Most existing architectures cannot deliver ISTORE's self-maintenance, primarily because they are based on general-purpose hardware and operating systems that must trade off reliability, performance, and focus for the ability to simultaneously support widely-varying hardware and application workloads. By focusing on servers customized to single applications, and by providing the framework for monitoring and adapting to environmental changes in an application-specific way, ISTORE allows for the construction of low-maintenance, highly available, high-performance systems.

The remainder of this paper describes the ISTORE architecture in greater detail. We begin in Section 2 with a description of ISTORE's modular, plug-and-play hardware architecture. Section 3 presents the architecture of the extensible runtime software layer that runs on top of that hardware and demonstrates how introspection provides adaptability and self-maintenance. Finally, we present related work in Section 4 and conclude in Section 5.

2 ISTORE hardware architecture

Attaining an introspective, self-maintaining system requires support from both hardware and software. Our proposed ISTORE hardware architecture reflects this combined requirement by providing both modular, flexible devices as well as dedicated, per-device processing to support the software infrastructure described in Section 3.

Traditional computer systems are built from a CPU-centric viewpoint: the CPUs sit atop a large hierarchy of busses, far away from the I/O devices attached to the bottom-level leaves of the hierarchy. This traditional configuration is not optimal for data-intensive, I/O-centric systems such as those built using the ISTORE architecture.

Thus, the ISTORE hardware architecture makes I/O devices first-class citizens by eliminating busses and attaching all components of the system directly to a high-bandwidth switched network. An ISTORE appliance is built out of physically interchangeable *device blocks* that plug into an intelligent *chassis*. A device block consists of one single-function device, such as a disk, combined with an embedded CPU and a network interface in a standard physical and electromechanical form factor. These device blocks fit into the bays of a chassis that provides uninterruptable power, cooling, environmental monitoring, and a scalable, high-bandwidth, redundant switched network.

A typical ISTORE appliance might be constructed out of several different, but physically interchangeable, types of device blocks. The exact configuration of blocks can be chosen to match the demands of the target application, providing easy hardware adaptability. For example, a database-focused appliance would be built primarily from disk blocks, but might also have memory blocks for caching. All appliances need one or more front-end interface/router blocks to provide scalable external connectivity to the system by translating between standard networks and protocols (e.g., ODBC over TCP over Ethernet) and internal application-specific protocols. One could imagine many other classes of device blocks that might be useful in constructing other types of appliances: tape blocks for backup, specialized media transcoding blocks for video servers or PDA proxies, high-performance CPU blocks for CPU-intensive applications such as scientific data processing, and a variety of front-end blocks that could be combined to provide multi-protocol access to the system.

ISTORE's LEGO-like approach to constructing a system out of single-function building blocks offers advantages in scalability, availability, and packaging. The system is inherently scalable, as devices connect directly to a switched network that offers scalable bandwidth and is capable of spanning multiple chassis, rather than to a fixed-capacity I/O bus. Additionally, the system can be incrementally scaled with heterogeneous hardware due to the modular, plug-and-play packaging of devices.

The network-based device interconnect enhances availability as well as scalability by replacing single-point-of-failure busses with redundant switched paths; device blocks can have multiple independent connections to the network to survive cable or network interface failure. Availability is also enhanced by having on-device intelligence, as devices can then autonomously check themselves, verifying correct operation of their integrated hardware and software via periodic scrubbing operations or "fire-drill" testing. Intelligent devices can also detect fatal errors or unexpected conditions that occur during normal operation and automatically disconnect themselves from the system, providing fail-fast behavior.

Finally, packaging all devices in small modular blocks with a common form factor has potential advantages in system packaging and operational cost. Certainly the packaging of an ISTORE system is more compact than that of a traditional SMP- or cluster-based server, thus saving machine-

room space, power, and cooling: a chassis the size of a single rack could hold over 100 3.5" device block bays with room to spare for network switching and UPSs.

3 ISTORE software architecture

The key features of ISTORE's hardware architecture are intelligent devices with dedicated, per-device processing power and a robust, scalable infrastructure that connects them. In this section we propose a software architecture for ISTORE that allows data-intensive applications to easily leverage these hardware features to provide scalable, self-maintaining services. This architecture has two main goals: first, the software system should provide the common mechanisms needed by all self-maintaining data-intensive applications. Second, it should provide these mechanisms in an extensible manner: the application should be able specialize the undifferentiated ISTORE software by customizing the supplied mechanisms and by providing application-specific interfaces to those mechanisms.

In this section we discuss a software architecture that we believe fulfils these goals. We focus first on the structure of the software system itself, and return to the extension techniques in Section 3.2.

3.1 Runtime software structure

The ISTORE runtime system is composed of three logical subparts: local system software components executing independently on each device block in the system, global system and application software components running in a distributed manner across all devices in the system, and front-end application components running on the interface/router device blocks.

The local software components that run on each intelligent device consist of the basic per-device OS services plus a set of local mechanisms. The OS services are comprised of a commodity embedded microkernel operating system plus a layer of standard microkernel modules that provide low-level process, communication, memory management, and device access primitives. We add an additional module that performs detailed, real-time monitoring of the operation of the device itself; this module monitors device access patterns, environmental parameters, indications of impending failure (such as repeated ECC failures on a disk), and utilization.

Above this layer of OS services is an extensible library layer that implements a set of basic non-distributed mechanisms. Most of these mechanisms are used to feed information to the global layer, or to perform local commands at the request of the global layer. One needed mechanism is a filter on the low-level device monitoring data that selects only the information or aggregate statistics appropriate for the application; for example, a transaction-processing database application might ignore the amount of disk bandwidth being used, and would select only information about the number of disk requests serviced per second. Another mechanism needed for all storage devices is a data access mechanism that controls data layout on the device and provides an application-specific interface to the data (such as a record-based interface for a database); this functionality might consist of two mechanisms, one for mapping application data quanta to disk objects, and another for mapping objects to raw device addresses). Other needed local mechanisms include device scheduling, caching, data tagging, and so on.

The ISTORE software architecture also includes a global software layer that interacts with the local layer. Since ISTORE has a shared-nothing hardware architecture, this global layer is actually a distributed program that runs on all of the device blocks in the system. The global layer consists of system-supplied extensible global mechanism libraries as well as the distributed application worker code that implements the particular network service for which the ISTORE has been specialized.

The mechanisms provided by the global software layer are primarily aggregation and control mechanisms that interact with the local mechanisms on each device. For example, an important global mechanism is monitoring. Unlike the local monitoring mechanisms associated with individual devices, the global monitoring mechanism can obtain information on all parts of the system, and can aggregate and interpret locally-generated data. For instance, it could compute average global utilization metrics that would allow it to decide if a particular device is being overutilized with respect to the rest of the devices in the system; this would be impossible at the local level since a device has no notion of a relative utilization scale. Global monitoring can also detect device failures that disable the local computation or connectivity on a device.

The global layer must provide several other key mechanisms, including a distributed directory service that tracks the location of data and metadata objects in the system, mechanisms for performing data migration and replication in response to hot spots and component failures, mechanisms for recovering from device failures and rebuilding redundancy of data that had been stored on the failed device, and mechanisms for integrating new components into the system. We present an example of how these components interact to provide self-maintenance in Section 3.3.

Finally, the ISTORE architecture includes a set of front-end-like application components that run on the interface/router device blocks in the system (in addition to the standard per-device-block system software layers). These extra application components are responsible for accepting requests from the LAN/WAN connection on the block and transforming them into invocations of the distributed application worker components that are part of the global software layer. For example, if an ISTORE system were being specialized to provide decision-support database service, the front-end block might receive SQL or ODBC queries across a LAN and perform query optimization locally; it would then invoke distributed relational operators (such as scans, sorts, and joins) running on the data storage devices to perform the data-intensive query execution.

3.2 Runtime system extensibility

While the local and global runtime libraries described in the previous section allow an appliance designer to directly access monitoring data and to manipulate low-level system state, those libraries are partitioned on functional boundaries that isolate the implementation of one mechanism from that of another. In contrast, appliance designers view their system at a higher level, thinking about it in terms of properties they wish the overall system to exhibit and the interfaces between their application and the base runtime system mechanisms.

To this end, ISTORE proposes an extension scheme in which the customizations to the base libraries are written using *domain-specific languages* (DSLs). The ISTORE DSLs are

specialized languages that allow the declarative specification of both the high-level behavior of particular aspects of the system and the application-specific interfaces to coordinated base mechanisms that implement that behavior. Each DSL in the ISTORE system encapsulates the high-level semantics of one logical component of the runtime system. By "logical component" we mean an overall behavior of the system, which might involve several different mechanisms of the runtime system. For example, one DSL might be used to express application-specific availability requirements. Because implementing a particular data availability strategy requires coordinating and directing a wide range of base mechanisms from replication and caching policies to device failure handling, the DSL compiler must translate a high-level description of the availability strategy into code that coordinates the relevant base mechanisms and presents the specified application interface. A similar specification process takes place for each extension to the base library that a programmer wishes to utilize; these specifications are compiled together with the local and global base mechanisms to produce the distributed runtime system through which the service application code interacts with the system.

When an undifferentiated ISTORE system boots, each device block contacts the system boot server (a special device block with removable media) and downloads both the customized mechanism libraries and the application worker code binaries. Although the extended libraries are loaded with the application when the system boots, they are structured as dynamic shared libraries so that they can be upgraded without requiring the application service to be restarted or the system rebooted.

We believe that ISTORE's DSLs are essential to exposing the power of ISTORE's introspective capabilities in a way that makes effective, application-specific utilization of those capabilities tractable. The programmer simply describes trigger values of the monitored system variables and the actions that should be invoked, and the DSL compiler decides how to tie the appropriate local and global monitoring mechanisms to appropriate low-level system actions. This allows individuals writing appliance applications to create a customized, adaptive runtime system that defines the appliance's low-level operation and the runtime system's interface to higher levels of the system.

In addition to providing programmability advantages by capturing precisely the relevant customizable features of a particular aspect of the system's behavior, DSLs offer advantages over low-level systems programming languages in terms of reliability, verifiability, and safety. While it is impossible to guarantee that the designer's program implements the desired algorithm, the high-level semantics of a DSL make it much more likely that the appliance designer's program is correct. For example, proper synchronization operations can be added by the DSL compiler to serialize accesses to a distributed data structure. Also, the DSL supports types that are natural to the module being designed, so type-checking in the compiler can detect many errors specific to the domain. Once the DSL compiler writer is satisfied that the DSL compiler is correct, designers inherit that verification effort. In terms of verifying the designer's DSL program itself, the high-level, constrained semantics of DSLs enable a more abstract form of semantic checking than is possible for a general-purpose programming language. Moreover, whole classes of general programming errors that are possible in low-level languages are not possible in well-designed high-level languages that hide details such as runtime memory management or that automatically add code to synchronize accesses to global data structures.

Because ISTORE proposes to use a DSL compiler to generate its runtime system, the DSL compiler also serves as a natural point in the system to add artificial diversity in the implementation of system components. Applying concepts from recent research into "survivable systems" that can continue operation in the face of internal bugs or malicious attack, we propose the use of software diversity to create multiple implementations of runtime system components [9]. Each implementation might be slightly different with respect to runtime memory layout, code ordering and layout, and system resource usage, within the constraints of the designer's DSL-based specification. Multiple implementations of a runtime system component could run simultaneously on different data replicas, checking each other with respect to high-level behavior.

3.3 From introspection to adaptation

Throughout this paper, we have claimed that intelligent hardware components, continuous monitoring, and extensible application-tailored software layers combine to produce adaptive, self-maintaining systems. In this section we provide an example of how this happens. Although this discussion applies to any data-intensive service application, we will use the example of a database to provide concreteness.

Consider the scenario of a slowly-failing data disk in a large system. With a traditional server architecture, the only report of this forthcoming failure would be the presence of media or ECC errors in the system log. If the system administrator were not watching the log (manually or automatically) at the time these errors occur, the disk failure might go unnoticed; even worse, the disk might return corrupt data, thereby driving the system into an inconsistent state.

In an ISTORE system, the microkernel monitoring module running on the local processor in the disk hardware block is responsible for monitoring the disk's health. When this module detects ECC failures or media errors on the disk, or even increased rates of ECC retries, it notifies the global fault-handling mechanism that its associated device may be about to fail. This mechanism is then responsible for migrating load off of that device, rescuing any unreplicated data, and rebuilding data redundancy. All of these actions are application-specific, and their exact behavior depends on application-specified policy.

In our database example, the fault-handling mechanism would first modify the global directory to remove the entries corresponding to the failing component's data; this prevents the system from sending more work to the failed device. Because database queries are typically transactional, the fault-handling mechanism might just discard the work currently in progress on the failing device and reissue those transactions to another replica of the data in the system. In a non-transactional system without coherent replicas, it might be necessary to checkpoint the computation and restore it on another replica of the data. At this point, the global fault handler could instruct the local software on the disk to shut itself down, effectively disconnecting the disk from the system and providing failstop behavior. Finally, the global fault handler would invoke the

data replication mechanism in order to rebuild the redundancy that was lost when the component failed. This action would in turn cascade through the layers of the system, using application-specified global layout policies to select another disk with spare space to hold the replica, global directory operations to record the location of the new replica, local mechanisms to copy the data, and local data layout mechanisms and policies to optimize the placement of the replica on the new disk.

In ISTORE, this entire process would take place automatically; when it was complete, the system would be in the same working order as when the failure began, and, because the system dynamically rebuilds redundancy, it would be as fault-tolerant as it was before the failure (assuming enough spare space is available). The failed component could then be replaced during regularly-scheduled maintenance, rather than requiring immediate human intervention. Finally, note that similar mechanisms would be used for automating performance-related administrative tasks, such as detecting a hot database table and automatically replicating its data.

4 Related work

The goals of the ISTORE system and the system-building techniques used to achieve these goals draw from research in a number of areas, including specialized appliances, extensible operating systems, domain-specific languages, plug-and-play hardware, and adaptive systems.

A specialized ISTORE appliance is similar to a class of single-function network servers that has recently emerged as a way to cost-effectively provide network services [13][24]. The device from Network Appliance is designed to be factory-configured to serve as either a file server appliance or a web server appliance by selecting the external protocol installed on the system; both appliances share a common hardware and system software platform. Although ISTORE is designed around a common hardware/software substrate as well, it addresses the higher-level goal of providing a generic meta-appliance that can be customized at all levels for a particular application.

Much of ISTORE's power and flexibility comes from its extensible runtime system, which leverages concepts from recent work in extensible operating systems [2][14][18][22]. As with extensible operating systems, the argument for extensible runtime systems is increased application performance and flexibility. But by addressing a higher level of the system, targeting the runtime system for a single-function appliance, and constraining extensions to those that can be expressed using DSLs, ISTORE sidesteps many of the thorny protection, security, and fairness issues found in designing an extensible operating system.

ISTORE's focus on domain-specific languages as an extension technique builds on past work in which DSLs have been used to specify cache coherence protocols and network protocols [4][17], or operating system extensions [14]. Like the OGI "microlanguages" project [20], ISTORE uses DSL-based specifications to generate system code; however, ISTORE will also be able to translate desired *global* system properties into appropriate use of *multiple* low-level mechanism libraries. In this sense the runtime system extensions in ISTORE borrow concepts from aspect-oriented programming [16].

ISTORE's LEGO-like hardware architecture extends several areas of previous work. While ISTORE and PC "plug-and-play" hardware both include ways to detect and identify new devices without the need for manual configuration, ISTORE also provides mechanisms by which the system can automatically begin using the capabilities of a newly-attached device. ISTORE's device blocks are a generalization of recent work in intelligent disks [1][15][21] and intelligent network interfaces [8]; ISTORE goes beyond these projects by providing a way to integrate multiple intelligent devices into a single hardware and software system designed to meet application-specific needs. Sun's Jini system [25] provides mechanisms by which intelligent devices can communicate and recognize each other and, as such, Jini fulfills a role similar to that of the low-level communication layers of ISTORE.

ISTORE's adaptive nature and self-tuning properties reflect a general trend in the software community. Researchers have proposed feedback-driven adaptation for extensible operating systems [23], databases [5][6], global operating systems [7], and storage devices [3][27]. The ISTORE architecture differentiates itself from these projects in two ways. First, ISTORE is a combined hardware/software architecture that integrates continuous, detailed monitoring at all levels of the system rather than retrofitting it into an existing system behind restricted interfaces. Second, an ISTORE-based appliance's adaptability is under complete control of the application via the DSL-based extension mechanisms, as opposed to being a static policy chosen by the system designer.

5 Conclusions

The growing importance of data-intensive network services demands new architectures for building the servers that support these services. We believe that specialized, optimized appliances built on top of intelligent hardware and introspective software form the right platform for data-intensive information services, as they provide the automatic self-administration, high availability, and performance needed by these services. Our approach stands in contrast to existing server architectures that are built from general-purpose hardware and system software, as such architectures are constrained by generic interfaces and abstraction barriers that make monitoring difficult and adaptation challenging.

By combining intelligent components with an extensible, reactive runtime system, the ISTORE architecture provides a powerful introspective framework for building the appliances needed to provide tomorrow's network services. Its modular intelligent hardware is adaptable, easily scaled, and reliable, while its extensible runtime system provides the base mechanisms needed by self-maintaining appliances yet can be easily customized by application designers to implement precisely the policies and semantics needed by their applications.

6 References

[1] A. Acharya, M. Uysal, and J. Saltz. "Active disks: programming model, algorithms and evaluation," in *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, Oct. 1998, pp. 81–91.

[2] B. Bershad, S. Savage, et al. "Extensibility, Safety, and Performance in the SPIN Operating System," in *Proceed-*

ings of the Fifteenth ACM Symposium on Operating System Principles (SOSP-15), Copper Mountain, CO., pp 267–284.

[3] E. Borowsky, R. Golding et al. "Eliminating Storage Headaches through Self-management," in *1996 OSDI Symposium*, Seattle, WA, Oct. 1996.

[4] S. Chandra, J. R. Larus, M. Dahlin, et al. "Experience with a Language for Writing Coherence Protocols," in *Usenix Conference on Domain-Specific Languages (DSL),* Santa Barbara, CA., Oct. 1998.

[5] S. Chaudhuri and V. Narasayya. "AutoAdmin 'What-If' Index Analysis Utility," in *Proceedings of ACM SIGMOD*, Seattle, 1998.

[6] S. Chaudhuri and V. Narasayya. "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server," in *Proc. 23rd Intl. Conf. on Very Large Databases (VLDB97)*, Athens, Greece, 1997, pp. 146-155, 1997.

[7] R. Draves, W. Bolosky et al. "Operating system directions for the next millennium," in *Proc. Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, May, 1997.

[8] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. "SPINE: A Safe Programmable and Integrated Network Environment," in *Proc. of the Eight ACM SIGOPS European Workshop*, September 1998.

[9] S. Forrest, A. Somayaji, and D. Ackley. "Building diverse computer systems," in *Proc. of the Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.

[10] Forrester. `http://www.forrester.com/research/cs/1995-ao/jan95csp.html.`

[11] Gartner. http://www.gartner.com/hcigdist.htm

[12] J. Gray. "Locally served network computers," Microsoft Research white paper, February 1995, available from `http://research.microsoft.com/~gray.`

[13] D. Hitz. "An NFS File Server Appliance," *Network Appliance, Inc., Technical Report 3001,* 1995.

[14] M. F. Kaashoek, D. Engler, et al. "Application Performance and Flexibility on Exokernel Systems," *Proc. 16th Symposium on Operating System Principles*, St. Malo, France, 1997.

[15] K. Keeton, D. A. Patterson and J. M. Hellerstein. "The case for intelligent disks (IDISKs)," *SIGMOD Record*, Vol. 27, No. 3, September 1998, pp. 42–52.

[16] G. Kiczales, J. Lamping, et al. "Aspect-Oriented Programming," in *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997.

[17] E. Kohler and M. F. Kaashoek. "A readable TCP in the Prolac protocol language," *submitted to ACM SIGCOMM '98.*

[18] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, and T. A. Proebsting. "Scout: A Communications-Oriented Operating System," in *HotOS-V,* May 1995.

[19] G. Papadopoulos. "Moore's Law Ain't Good Enough," Keynote address at *Hot Chips X*, August 1998.

[20] C. Pu et al. "The Microlanguage Project Overview," `http://www.cse.ogi.edu/DISC/projects/microlanguage/overview.html`

[21] E. Riedel, G. Gibson, and C. Faloutsos. "Active Storage For Large-Scale Data Mining and Multimedia," *Proceedings of the 24th International Conference on Very Large Databases (VLDB '98)*, August 1998.

[22] M. Seltzer, Y. Endo, C. Smith, K. Smith. "Dealing with Disaster: Surviving Misbehaved Kernel Extensions," in *Proceedings of the 1996 Symposium on Operating System OSDI II).*

[23] M. Seltzer and C. Small. "Self-Monitoring and Self-Adapting Systems," in *Proc. of the 1997 Workshop on Hot Topics on Operating Systems*, Chatham, MA, May 1997.

[24] SNAP!Server. `http://www.snapserver.com.`

[25] J. Waldo. "Jini Architecture Overview," *Sun Microsystems White Paper*, 1998.

[26] J. Wilkes. Personal communication, October 1998.

[27] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. "The HP AutoRAID Hierarchical Storage System," *ACM TOCS* 14(1):108–136, February 1996.