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

Abstract

Today's fast-growing data-intensive network services place heavy demands on the backend servers that support them. This paper introduces ISTORE, a novel server architecture that couples LEGO-like plug-and-play hardware with a generic framework for constructing adaptive software that leverages continuous self-monitoring. ISTORE exploits introspection *to provide high availability, performance, and scalability while drastically reducing the cost and complexity of administration. An ISTORE-based server monitors and adapts 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 an extensible software framework that allows the target application to specify monitoring and adaptation policies to the system.*

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 surpassed raw computational horsepower as the important commodity, with total growth in online data storage more than doubling every nine months [14]. 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 meet 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 millions of clients, and scalability that keeps up with 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, amounting to anywhere from two to twelve times the cost of the hardware [8][9][10][19]. 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 needs of the providers of new data-intensive services, and in particular to the need for servers that overcome the scalability and manageability problems of existing solutions. We argue that continuous monitoring and adaptation, or *introspection*, is a crucial component of such servers. Introspective systems are naturally self-administering, as they can monitor for exceptional situations or changes in their imposed workload and immediately react to maintain availability and performance goals. For example, an introspective system can track data access patterns and monitor media integrity, predicting and responding to data hot-spots or device failures by migrating and replicating data. As a result of these and similar mechanisms, human maintenance of an introspective 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 and incorporation of new hardware, are handled automatically.

This paper presents the ISTORE intelligent storage architecture, a hardware/software framework that simplifies the construction of next-generation introspective servers. The ISTORE framework can be flexibly customized to create a single-purpose server that is uniquely adapted to its application niche. Because ISTORE-based servers provide exactly one application or service, they 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 the application-appropriate semantics, optimizations, and responses to external stimuli needed for an introspective system.

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 hotswapped into a scalable, fully redundant hardware backplane. Each of the hardware "bricks" 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.

On top of this hardware platform, ISTORE provides a comprehensive software framework for constructing adaptive software that leverages continuous self-monitoring. The ISTORE runtime system automates the collection of monitoring data from the hardware bricks and provides applications with customized, application-specific views of that data; it further allows applications to define triggers over those views that automatically invoke adaptation routines when application-specific conditions are met. For common adaptation goals, the ISTORE system software goes further by providing an extensible mechanism for automatically generating monitoring and adaptation code based on application policy specifications expressed as constraints in declarative, domain-specific languages. Through this mechanism, an application designer can obtain the full benefits of an application-tailored introspective runtime system without having to write large amounts of code and without having to resort to ad-hoc techniques.

ISTORE's introspective nature sets it apart from today's existing general-purpose server architectures. 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, scalable, high-performance systems.

The remainder of this paper describes the proposed 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 illustrates the framework it provides for building introspective software. 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 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-based server is built out of physically interchangeable *device bricks* that plug into an intelligent *chassis*. A device brick 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 bricks 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 system might be constructed out of several different, but physically interchangeable, types of device bricks. The exact configuration of bricks can be chosen to match the demands of the target application, providing easy hardware adaptability. For example, a database server would be built primarily from disk bricks, but might also have memory bricks for caching. All servers tailored for network service applications need one or more front-end interface/router bricks 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 bricks that might be useful in constructing other types of servers: tape bricks for backup, specialized media transcoding bricks for video servers or PDA proxies, high-performance CPU bricks for CPU-intensive applications such as scientific data processing, and a variety of front-end bricks that could be combined to provide multi-protocol access to the system.

Beyond the obvious packaging advantages, ISTORE's LEGO-like approach to constructing a system out of single-function building blocks offers advantages in scalability and availability. The system is inherently scalable, as devices connect directly to a switched network that offers scalable bandwidth, 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 bricks 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.

3. ISTORE software architecture

ISTORE's hardware platform provides a robust, scalable, and powerful foundation for data-intensive network service applications. In this section, we propose a software architecture for the ISTORE runtime system that complements the hardware by allowing these applications to achieve self-monitoring, adaptation, and ultimately self-maintenance. This architecture has two main goals: first, the software system should provide a framework that allows the system designer to easily specify custom monitoring requirements and define related adaptation code. Second, and more importantly, the system should be capable of automatically generating monitoring code and adaptation algorithms for those adaptation goals that are common to data-intensive network services. The remainder of this section describes a proposed software architecture that we believe fulfills these requirements.

3.1. A framework for monitoring and adaptation

Central to the process of providing introspection is the collection, aggregation, and processing of monitoring data. ISTORE simplifies this process by unifying all monitoring data in the system into the abstraction of a single dynamic, system-wide database, and by providing a set of powerful data manipulation primitives on top of that database.

Each hardware device in an ISTORE system is a source of monitoring data, be it static representations of the device's capabilities, dynamic statistics on the device's environment and general health, or dynamic data streams containing access patterns and performance information. As this monitoring data is gathered, it becomes part of the system database; the schema of this database is a unification of the schemas of each class of device in the system, schemas that are in turn defined by the types of data collected by the sensors and software on that device.

An adaptive application running on an ISTORE system is clearly not interested in all the raw monitoring data being emitted at high volume from every device in the system; each application needs to select a subset of the available data and integrate that data to produce application-specific performance, health, or other monitoring metrics. To this end, ISTORE supports the definition of database-style *views* over the system monitoring database, expressed to the system as queries in an extended SQL-like declarative language. Views reduce the task of defining application-specific monitoring to a simple process of writing a declarative specification of which raw monitoring data is interesting and how to combine it, a process that can be simplified further via the use of graphical tools for query definition.

Providing runtime support for adaptive applications requires more than just allowing them to specify views over monitoring data, however; the system must also provide a way for those applications to express their notion of "interesting" situations to the system, where an interesting situation is defined by some state of the monitoring data that requires reaction and adaptation (*e.g.*, an overutilized component or a device failure). ISTORE achieves this by allowing applications to specify *triggers*, predicates over their monitoring views that are satisfied when an interesting situation occurs. Like views, triggers are specified to the system in a declarative, SQL-like language. When a trigger's predicate is satisfied, the ISTORE runtime system upcalls to an application-specified handler that is responsible for implementing the application's adaptation policy for the event that caused the trigger to fire.

Thus, by using the system monitoring database and its associated views and triggers, an ISTORE application inherits a significant amount of the needed mechanism for providing introspection, allowing the application designer to focus on the application-specific logic of monitoring, rather than having to reinvent and debug the mechanisms for these tasks from the ground up. Furthermore, an application can easily redefine its monitoring views and adaptation triggers while the system is running. As a result, it is trivial for an ISTORE application to dynamically change the set of statistics that it is monitoring and the conditions to which it will react. This provides applications with an additional level of adaptation that would not be possible were the monitoring and adaptation mechanisms coded statically for each application, as it allows applications to adapt their monitoring and adaptation strategies themselves as their workloads and environments change. For example, a database application being used for a mixed workload of online transaction processing and batch decision support might monitor different I/O metrics (*e.g.*, IOPS *vs.* delivered bandwidth) depending on the instantaneous properties of the workload, but might share the same reaction code in both cases.

3.2. Automatic generation of introspection

The framework described in the previous section relieves some of the burden of the designer of an ISTORE-based introspective service by allowing the designer to focus on the application-specific logic for monitoring and adaptation rather than on the mechanics of collecting and processing monitoring data. However, the framework by itself does little to simplify the mechanics of adaptation, as the designer must still write the trigger-invoked reaction code manually. In some cases, it may be possible to reuse existing application code or hooks into the application to perform the reaction tasks (for example, database applications already have recovery facilities that can be invoked upon failures), but in most cases the designer's task would be greatly simplified if the system could take over and automate the mechanics of adaptation as well.

To this end, the ISTORE runtime software architecture provides a means of automatically generating monitoring and adaptation code for those adaptation goals that are common across data-intensive network service applications. These adaptation goals include the tasks required by any self-maintaining system: detecting and recovering from component failure, recognizing and quenching data hot-spots and load imbalance, incorporating new hardware resources to allow for scaling and upgrading, and so forth. Of course, each application will have slightly different policies as to the importance of each of these goals and how they are to be met. Thus, any attempt to automate adaptation in these areas must be extensible, so that application policy can influence the generated mechanisms.

In our proposed ISTORE software architecture, this extensible automatic adaptation is accomplished via a combination of built-in reaction algorithms and mechanism libraries, which are integrated with application-specific policy information and the runtime framework described in the previous section by a policy compiler. The reaction algorithms encapsulate generic mechanisms for adapting to certain common situations, such as reacting to component failure by rebuilding lost data redundancy; the mechanism libraries provide a base set of mechanisms (such as object-based file access, directory services, replication, transactions, and so forth) that are used by the reaction algorithms. The key piece of the system, however, is the policy compiler. The input to this compiler is a declarative specification of the application's adaptation requirements, expressed as database-like *integrity constraints* over a view of the system monitoring database. The compiler translates this specification into the views and triggers needed to detect violations of those constraints, as well as into *adaptation code templates* (based on built-in reaction algorithms) that provide a generic mechanism for restoring the system to a state in which the constraints are again met. The reason for generating templates is that, while the system will always generate *correct* code to restore the system after a constraint violation, it may not always produce the *optimal* code for a particular application. Thus, while the generated code should be suitable for most applications, the use of a template allows motivated application designers to modify the code to incorporate more application-specific optimizations, such as using more relaxed locking protocols when replicating weakly-consistent data to quench a hot spot or load imbalance.

We believe that ISTORE's ability to automatically generate monitoring and adaptation code is essential to exposing the power of ISTORE's introspective capabilities in a way that makes effective, application-specific utilization of those capabilities tractable. The use of domain-specific declarative policy statements expressed as database constraints allows an application designer to specify precisely the relevant application-specific adaptation policies, without having to explicitly define monitoring procedures or coordinate adaptation mechanisms. The use of adaptation code templates preserves the possibility of further application-specific extension and optimization, while not requiring it in the common case.

Additionally, placing the burden of code generation primarily on the policy compiler simplifies the application designer's task beyond just reducing the amount of code that needs to be written: since the policy compiler (rather than a human) is responsible for generating adaptation code, it is much more likely that the resulting code is correct. The compiler uses as its raw materials a set of built-in algorithms and mechanisms; once these have been verified and debugged, all ISTORE application designers can inherit that verification effort. Similarly, since the compiler takes responsibility for generating the necessary housekeeping code in the adaptation code templates, whole classes of programming errors (including synchronization and memory management bugs) can be avoided. Finally, the compiler is a natural coordination point at which conflicting constraints or policies can be detected, and where cross-policy optimizations can be performed.

3.3. A brief example

In order to make the ideas discussed in the previous two sections more concrete, we now describe a sample system policy, the automatically-generated view, trigger, and adaptation code template produced to enforce it, and how that adaptation goal naturally helps to provide self-maintenance. The particular adaptation goal we will examine is maintaining a fixed level of data availability by ensuring that there are always three replicas of all data objects in the system.

Given a declarative statement of this goal, the policy compiler first determines what status and performance information is required to detect a violation of the redundancy requirement. In this example, the requisite status information is the "health" of the disks in the system, *i.e.*, whether each disk is "dead" or "alive." The compiler then constructs a view by selecting the health field of the database for each entry that corresponds to a disk. Note that we are assuming the policy requires all three replicas to be stored on disk; the designer might instead specify that one of these replicas may be stored in volatile or nonvolatile memory, in which case the health status of those components would also become part of the view.

The next step is to define a trigger that detects when the "health" field of a record in the view changes to "dead" and that then invokes the appropriate adaptation code. To do this, the policy compiler specifies both the condition that will cause the trigger to fire and the relevant information from the database (defined as yet another view) that the adaptation code might use in reacting to the trigger. When the trigger fires, this second view is passed as an argument to the reaction code. In our example, the argument view might consist of the identities of the data objects that were stored on the dead disk and the static and dynamic utilization of all the disks in the system (*i.e.*, how full they are and how much demand has been placed on their seek capacity and transfer bandwidth over one or more preceding time periods).

Finally the policy compiler generates an adaptation code template. In our three-replica example, this template would implement a generic replication mechanism that, for each data object that was stored on the disk that died, finds one of the two remaining replicas of that object, locks the replica, performs a byte copy of the object to the least utilized disk in the system, unlocks the object, and updates the system's index of data object locations. This adaptation code can then be customized by the system designer to exploit application semantic information, if desired. For example, the system designer might want to store the new (third) replica of an object only on a disk not already holding a replica of that object, might want to employ an application-specific compression algorithm to compress all third replicas if disk space is becoming scarce, or might want to use an application-specific relaxed locking policy during the copy operation.

Thus the ISTORE software allows a designer to create what is essentially an application-specific introspective runtime system just by specifying the key runtime invariants for their application. Moreover, ISTORE's flexible declarative policy specification framework allows the easy construction of systems that are fully self-maintaining, *i.e.*, that not only performance-tune themselves in response to workload variations but also perform maintenance tasks like healing the system after a component failure. Our example illustrates this: when a disk dies this fact will be reflected as an update to the monitoring database which will fire the trigger that will in turn invoke the replication reaction code described above.

3.4. Hardware support for software introspection

ISTORE's ability to effectively support self-maintaining applications relies on the interplay of its intelligent hardware and introspective software framework. This integration of hardware and software is most evident in the implementation of the monitoring database. The adaptive software framework we have described assumes the information stored in the monitoring database is somehow continually updated. The source of these updates may take any number of forms, including simple hardware monitors (*e.g.*, temperature sensors or performance counters) or traditional software instrumentation for statistics gathering. But by using intelligent hardware, ISTORE can implement not only statistics gathering but also monitoring for arbitrary conditions directly on the hardware that is being monitored. As a result, the repository of monitoring data can be a virtual database, rather than a materialized one, with the hardware implementing views and triggers directly by filtering and reacting to data as it is gathered. Some triggers can be implemented on each brick in a purely local fashion, *e.g.,* "detect when utilization exceeds 90%." Others take the form of a distributed computation that synthesizes local monitoring data, *e.g.*, "detect when less than 10% of total system disk space is free." Finally, some triggers can be implemented on a single brick but require gathering indirect information about remote bricks, *e.g.,* "detect when another brick stops responding to network messages."

Another key advantage of ISTORE's union of intelligent hardware and introspective software becomes evident when systems of heterogeneous devices are considered. As ISTORE's monitoring database incorporates not only dynamic runtime status but also static information about the functional and performance capabilities of each hardware device, introspective application software can leverage ISTORE's self-characterizing hardware by extracting this static information from the database when making adaptation decisions. For example, the system might include disks with different seek latencies, and a system policy constraint might specify that randomly-accessed data should be kept on the disks with the smallest seek latencies while large sequentially-accessed objects may be stored on the disks with the longest seek latencies. Thus the ability of adaptation code to query ISTORE's database to help make appropriate adaptation decisions complements ISTORE's plug-and-play hardware framework by allowing the system's adaptation mechanisms to automatically and intelligently utilize new and heterogeneous hardware components.

4. Related work

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 [16], databases [4][5], global operating systems [6], and storage devices [3][20]. The ISTORE architecture differs from these projects in two ways. First, ISTORE is a combined hardware/software architecture that systematically integrates continuous, detailed monitoring at all levels of the system. Second, an ISTORE-based server's adaptability is completely controlled by the application's dynamic definition of views and triggers, not imposed statically by a system designer.

ISTORE's use of a database with views and triggers to manipulate monitoring data borrows heavily from the design of relational databases [2][18]. The idea of storing system statistics and monitoring data in a relational database similarly mirrors recent work in no-knobs database design [13], although ISTORE extends this work by incorporating dynamic access streams into the database and by using intelligent hardware to create a non-materialized distributed database.

ISTORE's device bricks are a generalization of recent work in intelligent disks [1][12][15] and intelligent network interfaces [7]; ISTORE goes beyond those projects by integrating multiple intelligent devices into a single hardware and software system designed to meet application-specific needs, and by using intelligence to enable adaptation, not just to execute application code.

Finally, a specialized ISTORE server is similar to a class of single-function network "appliances" that has recently emerged as a way to cost-effectively provide network services [11][17]. ISTORE goes beyond these appliances by addressing the higher-level goal of providing a generic framework for building such appliances, rather than focusing only on the optimizations needed for a particular application.

5. Conclusions

The growing importance of data-intensive network services demands new server architectures. We believe that specialized, optimized single-purpose servers that merge intelligent hardware and introspective software provide the automatic self-administration, high availability, scalability, and performance needed by these services. Our approach stands in contrast to existing server architectures that, because they are built from general-purpose hardware and system software, are constrained by generic interfaces and abstraction barriers that make it difficult to implement effective self-monitoring and adaptation.

By combining intelligent components with an extensible, reactive runtime system, the ISTORE architecture provides a powerful, flexible framework for building the introspective servers needed to support tomorrow's network services. ISTORE's modular intelligent hardware is adaptable, easily scaled, and reliable, while its runtime system simplifies and automates the creation of the application-specific monitoring and adaptation mechanisms that are essential to introspective, self-maintaining systems.

6. Acknowledgments

This work was supported by DARPA under grant DABT63-96-C-0056. The authors wish to thank Jim Beck, Joe Hellerstein, and Margo Seltzer for their insightful comments on earlier drafts of this paper.

7. 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*, Oct. 1998, pp. 81–91.

[2] M. Astrahan, et al., "System R: Relational Approach to Database Management," in *ACM Transactions on Database Systems*, 1(2): 97–137.

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

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

[5] 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.

[6] 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.

[7] 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.

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

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

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

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

[12] 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.

[13] S. Lakshmi, "No-Knobs Database Operation - An Informix CTO Initiative," *guest talk given as part of U.C. Berkeley CS294-2 class*, 20 November 1998.

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

[15] 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.

[16] 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.

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

[18] M. Stonebraker, et al., "The Design and Implementation of INGRES," *ACM Transactions on Database Systems*, 1(3): 189–222.

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

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