ROC-1: Hardware Support for Recovery-Oriented Computing

David Oppenheimer, Aaron Brown, James Beck, Daniel Hettena, Jon Kuroda, Noah Treuhaft, David A. Patterson, and Kathy Yelick

Computer Science Division, University of California at Berkeley

{davidopp,abrown,beck,danielh,jkuroda,treuhaft,pattrsn,yelick}@cs.berkeley.edu

Abstract

We introduce the ROC-1 hardware platform, a large-scale cluster system designed to provide high availability for Internet service applications. The ROC-1 prototype embodies our philosophy of Recovery-Oriented Computing (ROC) by emphasizing detection and recovery from the failures that inevitably occur in Internet service environments, rather than simple avoidance of such failures. ROC-1 promises greater availability than existing server systems by incorporating four techniques applied from the ground up to both hardware and software: redundancy and isolation, online self-testing and verification, support for problem diagnosis, and concern for human interaction with the system.

1 Introduction

In recent years the number and variety of Internet services has proliferated. The economic cost of downtime to service providers is substantial—millions of dollars per hour for brokerages and credit card companies, hundreds of thousands of dollars per hour for online retailers and services like amazon.com and eBay [34] [21]. Despite this pressing need for high availability, and many decades of research by the fault-tolerance community, existing techniques for avoiding failures have not prevented a string of costly and well-publicized problems for those who operate Internet services. Indeed, InternetWeek recently reported that 65% of the sites they surveyed had experienced at least one customer-visible outage during the previous six months, while 25% had experienced three or more [34].

We believe these problems stem from a fundamental mismatch between the goals and assumptions of much previous availability research, and the realities of modern Internet services. For example, much fault-tolerance research has focused on reducing the failure rate of individual system hardware and software components in isolation. Error-correcting memory reduces the failure rate of DRAM, and formal models of software allow specified failure modes to be checked and eliminated. But Internet services are created by composing a complex set of hardware and software components that are heterogeneous and subject to continuous upgrade, replacement, and scaling in their numbers. These systems are too complex to model formally, and reducing the failure rate of individual components may not substantially reduce the rate of overall system failures due to unexpected interactions between components. Moreover, a focus on reducing hardware and software failure rates ignores the fact that many system failures result from unavoidable human interaction with systems, *e.g.*, during hardware and software upgrades and repairs, problem diagnosis, configuration, system expansion, and the like. Indeed, the design of existing server systems generally overlooks the importance of human factors and the fact that humans are fallible [4].

Put simply, despite reductions in hardware and software failure rates, failures are inevitable. With this observation in mind, we propose *recovery-oriented computing*, a philosophy for achieving high availability that focuses on detecting and recovering from failures rather than preventing them entirely. Quantitatively one can view this philosophy as focusing on the reducing the contribution of Mean Time to Repair to overall system unavailability. We propose four techniques as the initial basis for ROC: redundancy and isolation; online self-testing and verification; support for problem diagnosis; and concern for human interaction with the system. The ROC philosophy was first introduced in [3] and its goals and techniques are described in more detail in [2].

In this paper we focus on ROC-1, a prototype hardware platform for Internet services that incorporates hardware support for mechanisms essential to Recovery Oriented Computing. The ROC-1 system is a 64-node cluster that holds 1.2 TB of online storage and that fits in three standard machine room racks. Each *brick* node, packaged in a half-height disk canister, consists of an

18 GB SCSI disk along with a custom-designed x86 PC board that contains a Pentium II mobile processor, DRAM, network interfaces, and a Motorola MC68376-based diagnostic processor connected to a private diagnostic network. The diagnostic processors, described in more detail in Section 3, collect data from environmental sensors distributed throughout the system and control fault injection hardware.

The remainder of this paper is organized as follows. In Section 2 we present an overview of Recovery Oriented Computing. Section 3 describes the ROC-1 hardware in detail, with a particular focus on its hardware mechanisms that support ROC. Section 4 describes the status of our software support for ROC and our evaluation methodology, Section 5 briefly discusses related work, and in Section 6 we conclude.

2 An overview of Recovery Oriented Computing

Traditional fault-tolerant computing research has generally focused on systems with software that evolves slowly if at all, that use a single well-defined hardware platform, that are operated by an extremely well-trained staff, and that are given workloads that can be well-defined at development time and that do not change during operation. In such an environment, techniques such as the use of redundant hardware with built-in error checking, extensive pre-deployment testing, and formal modeling can be applied to achieve high availability.

But Internet service systems are developed and deployed under significantly different conditions. For these systems, software and hardware vary widely and evolve during system operation; development cycles are short; workloads are constantly changing; testing is minimal; operators may be poorly trained; and software and hardware components are commodity, heterogeneous, and of a scale that defies simple modeling. As a result, traditional high-availability techniques have proven inadequate, too inflexible, too time-consuming, or too expensive for use in these systems. Therefore instead of trying to prevent failures, Recovery-Oriented Computing focuses on techniques to expose latent errors early, to contain failures when they do occur, to improve the reliability of error detection and recovery mechanisms, to help operators diagnose and repair problems, and to tolerate operator error during system maintenance tasks. The techniques we are implementing fall into four general categories: redundancy and isolation, online testing and verification, support for online problem diagnosis, and design for human interaction.

Redundancy dictates that the system should have extra software and hardware components, paths between them, and copies of data, so that there exists no single point of failure in the system. *Isolation* refers to the ability to partition the system so that one partition of components cannot affect any other partition. This property is useful for concealing the effects of failures, avoiding propagation of failures, allowing incremental system upgrade, permitting repair of broken components without taking the entire system down, and allowing testing and operator training on parts of a live system without affecting portions of the system not being used for testing or training.

A second principle of ROC is the use of *online testing and verification* to detect latent software and hardware errors, and to test the correctness of error-handling and recovery procedures. Shrinking software development cycles, ever-growing software feature sets, and a proliferation of hardware and software building blocks have made fully testing products—let alone the interactions of features within and between products and versions of products—nearly impossible [17]. A system with hardware and software isolation can be instrumented at its component interfaces to inject test inputs or faults and to observe the system's response.

Online testing and verification may prove particularly useful in addressing latent errors caused by faults. A *fault* is the original cause of an error, such as a programmer mistake or an alpha particle striking a memory cell. When a fault occurs, it creates a *latent error* that is said to become *effective* when it is activated (*e.g.*, when a faulty code path is executed or data is read from a corrupted memory cell). Studies of accidents in other fields, such as the nuclear power industry [27] [29], indicate that latent errors can accumulate and lead to multiple simultaneous failures. Self-testing makes it possible to find latent errors that may not be exposed during offline testing, hence extending the testing cycle throughout the lifetime of the system.

Just as some modern DRAM incorporates scrubbing logic to detect and correct latent single-bit errors using ECC information, online testing and verification can be used on a system-wide level to detect latent hardware and software errors before they turn into failures. Latent errors may manifest themselves during online testing because the system can test the full stack of applications, operating system, drivers, and hardware at the user's site; no vendor can test for all possible combinations of these factors. As two specific examples, we believe online testing can help significantly in exposing transient bugs that may only become manifest when certain sets of components interact, and can also help to reveal bugs in error0handling code. Error and fault handling code paths are generally difficult to test because of the infrequency with which errors are encountered in traditional "beta testing" scenarios and the difficulty of testing under all possible scenarios of product, version, and patches applied to software, firmware, and hardware.

Additionally, we propose the use of fault injection to reduce latent human errors. By continually and automatically training and retraining operators to diagnose and handle failure scenarios, a ROC system should experience a smaller Mean Time to Repair than a non-ROC system when human intervention is necessary to configure, upgrade, or repair the system.

Despite the proliferation of management standards and APIs such as SNMP [7], WBEM [10], and JMX [33], pinpointing the root cause of system failures and performance degradations remains challenging. We believe that like online testing and verification, *support for diagnosis* of system problems must be built into a system from the ground up. This is in contrast to systems that mask failures using redundancy; in those systems failures may manifest themselves as performance degradations as load is shifted onto non-failed system components without any indication of what has happened. Instead, we suggest that all components should have error-reporting interfaces and that failure information should be propagated up through all levels in the system so that the errors can be handled appropriately by an application or an administrator. Logging failures at all levels of the system allows both early detection of components that are on the verge of failure and eases post-mortem investigation of a failure that could not be prevented.

Finally, recovery-oriented computing recognizes that reducing human error is essential to improving system availability [4]. Because not all tasks of system operation can be automated, interaction of human operators with the system is inevitable. We believe large-scale server systems should be designed with *concern for human interaction* so as to minimize the likelihood and seriousness of these errors. Many of the techniques already discussed support this goal, *e.g.,* the ability to insert faults allows simulation of failures for operator training; comprehensive data gathering for problem diagnosis makes easier the task of finding the root cause of problems; and the use of redundancy and isolation allows failed components to remove themselves from the system for repair or replacement at the operator's leisure. We are currently investigating additional software techniques for reducing the impact of human error such as the ability to undo all human-initiated operations.

Overall, the goal of the UC Berkeley Recovery Oriented Computing project is to substantially improve system availability by applying techniques of the type just described. Although many of the techniques for ROC are software-based, specialized hardware support can provide additional benefits by implementing some of the techniques at a lower level than is possible in software. To this end we have built the ROC-1 hardware prototype, a prototype server platform for exploring the techniques of Recovery Oriented Computing. We describe this system in the next section.

3 Hardware Overview

The ROC-1 hardware prototype is a 64-node cluster composed of custom-built nodes, called *bricks*. Figure 1 shows the organization of a brick. Each brick contains a 266 MHz mobile Pentium II processor, an 18 GB SCSI disk, 256 MB of ECC DRAM, four redundant 100 Mb/s network interfaces connected to a system-wide Ethernet, and an 18 MHz Motorola MC68376-based diagnostic processor (DP) connected to a private diagnostic network. For both space and power efficiency, the bricks are each packaged in a single half-height disk canister. A photograph of a brick partially inserted into its canister appears in Figure 2.

The ROC-1 bricks are fully interconnected via redundant Ethernet networks. Each of the four brick network interfaces is connected to one of sixteen first-level network switches. Each of these switches connects via a Gigabit Ethernet uplink to one of two ganged Gigabit switches, each of which is capable of routing the nodes' full crossbar bandwidth. Figure 3 shows the interconnection topology of the network connecting the nodes. Although we considered using System Area Network (SAN) technology when designing ROC-1, the availability of network interface parts at

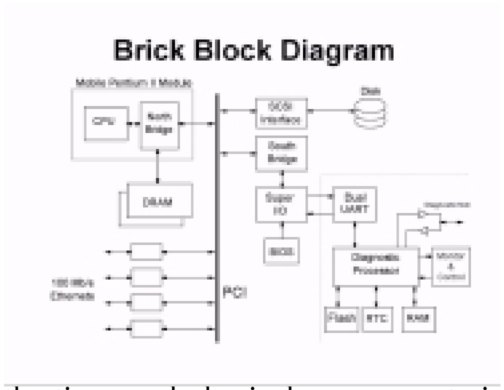

Figure 1: Block diagram of a ROC-1 brick board.

the time, and physical space constraints on the processor board, led Gigabit Ethernet to be the only viable networking option.

Though logically structured as a cluster, ROC-1 is designed with a processor-to-disk ratio higher than that used by most existing server clusters. This choice was made to ensure that sufficient excess computing capacity is available, beyond that required by the application software running on each node, to dedicate to self-monitoring and other self-maintenance functionality.

An additional important difference between ROC-1 nodes and standard server nodes is ROC-1's incorporation of a diagnostic subsystem. The per-brick diagnostic processor is a prototype for what in a production system would be a small, independent, trusted piece of hardware running well-verified monitoring and control software. The diagnostic subsystem consists of a Motorola 68376 CPU that includes a Controller Area Network (CAN) [19] bus controller, a dual UART that connects the 68376 to two serial port interfaces on the PC chipset, 1MB of FLASH RAM, and 1MB of battery-backed SRAM. The diagnostic subsystem's hardware monitoring functionality is fed by an array of sensors on each brick and on the system backplane. Each brick contains three thermometers, four voltage monitors, a sensor that detects attempts to remove the brick from the system, a battery warning for the diagnostic processor's SRAM memory, and an accelerometer on the disk drive to detect abnormal vibrations due to improper disk mounting or even earthquakes.

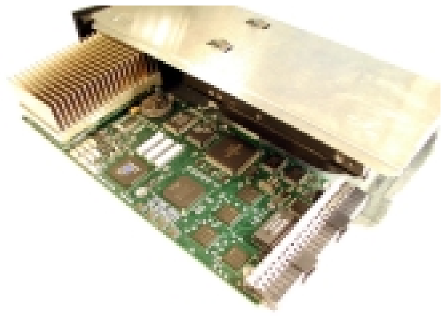

Figure 2: A ROC-1 brick. The processor board is shown partially removed from the canister, and with its disk removed.

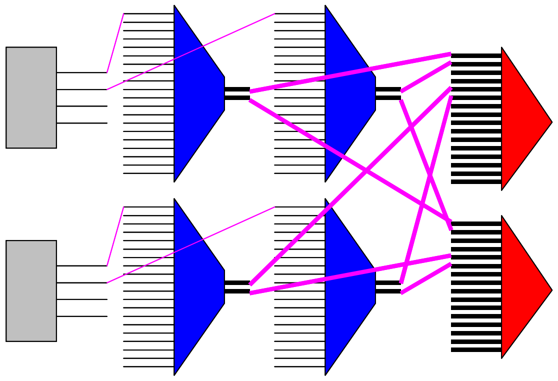

Figure 3: ROC-1's fault-tolerant routing topology. Each square is a node, each trapezoid is a second-level 100 Mb/sec switch with 1 Gb/sec uplink and each triangle is a first-level Gigabit Ethernet switch. There are four paths from each brick to a distinct second-level switch, and two paths from each second-level switch to a distinct first-level switch. As a result, there are two completely distinct routes from each node to each other node.

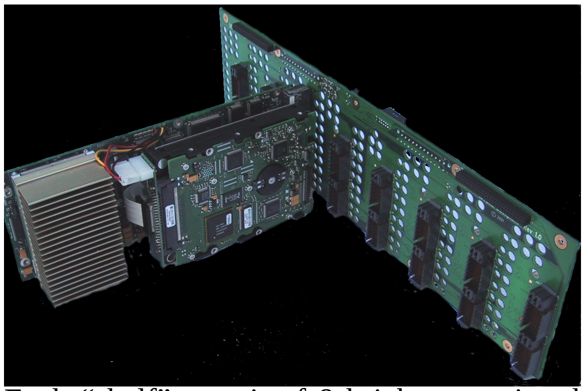

Figure 4: A ROC-1 brick connected to the backplane for one shelf. Each brick connects to the backplane via two 72-pin connectors and two larger power connectors. The 72-pin connectors contain wires for four full-duplex 100 Mbps Ethernets and one full-duplex serial line.

Each "shelf", a unit of 8 bricks, contains three more thermometers, a fan stall detector, and a humidity sensor.

The control functionality of the diagnostic subsystem allows power to the disk, Pentium II CPU, and network interfaces to be independently turned on and off; turning off power is one way to simulate component failure. The DP also controls extra hardware that allows faults to be injected onto the brick's SCSI and memory buses and the brick's Ethernet transmit and receive lines. Finally, the DP can reset the Pentium II CPU. Monitoring and control data is exchanged among diagnostic processors through a private diagnostic network built from per-shelf 1 Mb/sec

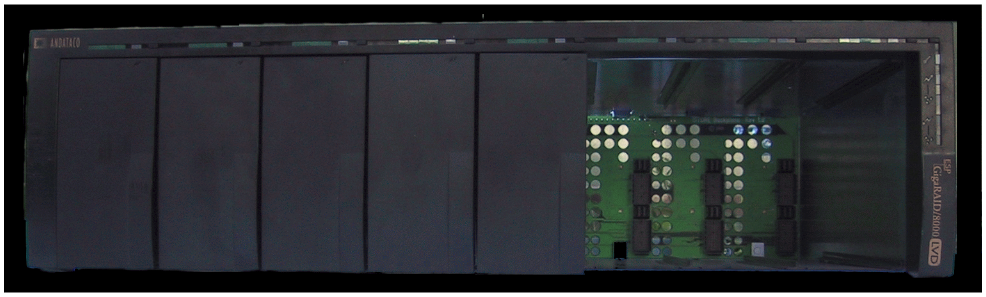
Figure 5: A shelf with five bricks inserted and three empty spaces.

CAN buses. A photograph of a ROC-1 brick connected to the backplane appears in Figure 4, and a photograph of one ROC-1 shelf appears in Figure 5.

Although designed primarily with availability and maintainability in mind, ROC-1 also offers good cost-performance as a Internet server. Internet servers are typically deployed at colocation sites, which charge on the order of $1000 per rack per month and $200 per month for extra 20 Amp power circuits. Thus physical space and power consumption should be minimized when designing Internet servers. To address space efficiency, ROC-1's packaging one disk with one CPU in a half-height disk canister allows eight nodes to fit in 3U of rack space. Power efficiency was also considered: by using a ratio of one disk for each CPU, each node needs only the processing power of a low-cost, low-power, embedded CPU. This contrasts with traditional server systems which attach multiple disks to each node and use one or more expensive, high-power, server CPUs. We have measured average brick power consumption at 27.6W and peak brick power consumption at 50W, for a total power consumption per shelf of 221W average and 400W peak. Our first-level network switches add 6.6 kW and our second-level network switches add 5.3 kW, for a total power budget of almost 12 kW just for network switches. Including network switches, then, total peak system power consumption for 64 nodes is 15kW. Thus while our nodes are quite power-efficient, for a non-prototype ROC-1 system to make sense, newer, lower-power switches should be used.

In addition to saving cost by using an embedded CPU, packing a CPU with each disk amortizes the cost of disk enclosures, power supply, cabling, and cooling that are already needed for disks (and are provided by existing disk enclosures) across both the disks and CPUs. Finally, from a performance standpoint, by using only one disk on each I/O bus, the system eliminates bottlenecks associated with placing multiple I/O devices on a bus, *e.g.*, bus contention during arbitration and data transfer. Note that although the current ROC-1 system occupies three machine room racks, many of the largest components such as UPSes and network switches were purchased several years ago. We believe that by using today's smaller UPSes and switches the system could be made to fit in approximately two racks.

The remainder of this section focuses on specific characteristics of the ROC-1 hardware architecture intended to support the four major principles of ROC: isolation and redundancy; online self-testing and verification; support for problem diagnosis; and concern for human interaction with the system.

3.1 Redundancy and isolation

The ROC-1 hardware platform implements the classic principle of avoiding any single point of failure by incorporating redundant network interfaces, network switches, shelf power supplies, and fans; using an uninterruptable power supply; and using a diagnostic processor and diagnostic network that provide a secondary path to control each node. This last feature is useful in that it allows bricks to be power-cycled when all of a node's Ethernet interfaces fail or a design flaw in the Ethernet driver software or firmware has made a node unable to communicate via the Ethernet network. This hardware support for redundancy is orthogonal to higher-level redundancy provided through software techniques such as replicated storage of data or use of redundant front-end processes to handle interaction with user clients.

Related to redundancy is isolation—the ability to partition the system logically as well as physically during normal operation. Logical partitioning is provided through standard mechanisms such as running system software components in different virtual machines or different operating system address spaces protected by memory management hardware, while physical partitioning is provided by the ability of the diagnostic processor to control power to each network interface and CPU, and each disk. The ability to turn off network interfaces allows a node to be physically isolated from the rest of the system at a very low level but left online and accessible through the diagnostic network for debugging purposes.

As mentioned in Section 2, partitioning is useful for a number of reasons: it helps to confine faults to the software or hardware module in which they occur; it allows failed system components to be diagnosed and repaired online without affecting the rest of the system; it enables incremental online system upgrade and scaling; and it allows operators to experiment and train using the actual, deployed hardware and software configuration at a site without affecting normal system operation.

3.2 Online testing and verification

Online testing analyzes the exact hardware and software configuration used at a particular server site. An isolated partition of the system with redundant copies of the actual dataset of a site's application can be established; the actual request load to which the system is exposed during normal operation can then be mirrored during the test from the load sent to the "production" partition of the system, or a historical trace of previous requests can be used as the test load.

We foresee three types of online testing: correctness tests, robustness tests, and operator tests. In correctness tests, an input or fault with a known expected response by the system is inserted, and hardware or software monitoring checks for the expected response. In robustness tests, a randomly generated input or fault is inserted, and software checks to make sure that the system fails in a reasonable way. In operator tests, a failure is caused that requires a human operator’s response, and the system checks to make sure the problem is corrected properly and within a desired time period. Note that of these three types of tests, only correctness tests require foreknowledge of the correct response to the test; robustness and operator tests only require checking that the state of the system after the test is sane.

At the hardware level, *in situ* testing is enabled in ROC-1 by the ability to isolate nodes at the Ethernet level and by the low-level fault injection hardware integrated into each brick. For example, the diagnostic processor can inject faults into the memory bus and the transmit and receive lines on the network interface in order to simulates errors such as bit flips. As a substitute for specialized hardware support, we are also investigating the possibility of using software-based virtual machine technology to enable fault insertion on commodity hardware platforms.

3.3 Support for online problem diagnosis

At the hardware level, online problem diagnosis is supported by the environmental sensors on the bricks and the system backplane. Each brick contains three thermometers (for detecting components that are beginning to overheat), four voltage monitors (for detecting power irregularities), a sensor that detects attempts to remove a brick from the system (so that critical state in the node can be saved before power is removed), a monitor that indicates when the diagnostic processor's SRAM memory battery is low, and an accelerometer on the disk drive to detect abnormal vibrations such as those that might cause a head crash or that might occur during an earthquake. Furthermore, each shelf contains three thermometers and a humidity sensor for detecting such conditions as overheating or flooding, and a fan stall detector for detecting failure of the power supply fans. Though the software for automatically processing the data received from these sensors is still under development, we intend that the information obtained from them is analyzed by the diagnostic processor for automatic reaction, or is sent to a system administrator's diagnostic console. As an example of automatic reaction, a diagnostic processor that detects the ambient temperature rising excessively might throttle back the brick's CPU speed, power off the brick's disk, or power down the brick entirely, in order to reduce power consumption and hence decrease the amount of heat produced by the node.

Recognizing that growth and upgrade can lead to an actual system configuration that differs significantly from the original design, and that therefore an operators' understanding of a system's configuration may not match the actual configuration, ROC-1 incorporates automatic network and power topology self-discovery. Network configuration self-discovery can be accomplished using standard SNMP-based network mapping tools, while power topology can be mapped using powerline networking products that utilize power circuits for communication between PCs [35].

Finally, we intend to use the diagnostic processor's Flash RAM as a "flight recorder," storing system logs and hardware status information in nonvolatile memory to enable post-mortem failure investigation. Flash RAM is particularly well-suited to this task because critical data can be written to it as a node is failing in less time than is needed to write to a disk, and because data can be written to the Flash RAM even in the case of a disk failure.

3.4 Design for human interaction

Because operator failure is a significant cause of downtime, ROC-1's packaging is designed with maintainability in mind: the only field-replaceable unit is the brick, which is as easy for an operator to replace as a disk in a standard RAID array. ROC-1's modular hardware architecture is homogenous and free of cables within a shelf, thus simplifying upgrade and replacement. Finally, the isolation and fault-injection techniques described earlier can be combined to train operators—

a portion of the system can be isolated and failures can be injected into the actual system hardware, software, and workload request stream to allow operators to practice responding to problems.

4 Applications and evaluation

While the ROC-1 hardware has been built, its software is still under development. Rather than write a generic software layer implementing all of the ROC techniques, we are starting by integrating some of the ROC techniques into NinjaMail, an electronic mail system developed as part of the UC Berkeley Ninja project [15]. In particular, we are implementing undo for administrative actions and online testing of modules. Our initial email workloads are based on the SPECmail [31] benchmark and mirroring the email load applied to the UC Berkeley EECS department's email server. To provide an initial fault workload we are instrumenting the Java Runtime Environment in which NinjaMail runs, to return errors from I/O operations.

Evaluating the effectiveness of techniques aimed at improving system availability is inherently more difficult than evaluating those that improve performance, the traditional metric of systems research. This is because measuring availability requires not only an application workload, as for performance benchmarks, but also a fault workload that simulates the faults to which a real system is expected to be exposed during operation. Establishing a fault workload requires defining a fault model (*i.e.*, enumerating the type and frequency of faults to which the system will be exposed) and implementing mechanisms for injecting these faults into the system under evaluation. Our approach to benchmarking availability, developed in earlier work, is to measure variations in Quality of Service metrics, *e.g.*, throughput, latency, completeness of results returned, and accuracy of results returned, as a fault workload is applied to a system running a standard performance benchmark [5]. Because our first application is email, we intend to use the SPECmail benchmark as the performance benchmark, and we will measure such metrics as throughput, error rate, and number and impact of human errors made in administering the service.

A preliminary evaluation of the ROC-1 prototype indicates that it does well in achieving the goals of enforcing isolation and redundancy, allowing low-level system monitoring, and enabling hardware fault injection. The per-node power consumption is also quite reasonable: the average power consumption of 28W/node is better than the typical 1U server consumption of 76W/node but not quite as good as some recent power-optimized systems that consume 15W/node [30]. Overall system power consumption in ROC-1 could be significantly reduced by using new lower-power switches—the switches we used consume five times as much power as all the brick nodes combined.

5 Related Work

The notion of designing a system that assumes code will be buggy and therefore uses techniques for fast recovery is reflected in recent work on the design of Internet services that are partitioned into stateless "worker modules" and stateful back-end nodes [11] [26]. The worker modules in these systems operate purely from soft state, and therefore can be restarted without damaging or losing system data. Moreover, because they do not keep their own persistent state, worker modules do not incur startup time overhead in recovering lost data or restoring data consistency when they are restarted. Other recent work has focused on formalizing the properties of such restartable systems and on determining how and when they should be rebooted [6] [18].

Isolation and redundancy are traditional fault-tolerance techniques. Fail-fast system modules were originally identified as an important system structuring principle by Gray [14], and the recent use of shared-nothing hardware has enabled clusters to achieve fault containment for free as compared to shared-memory multiprocessors that must use virtual-machine techniques for such containment [13]. Redundancy via process pairs was also identified by Gray as an important technique [14], and data replication has long been used in RAID systems to enhance storage availability [8]. Our ROC philosophy is novel not in its use of these techniques for enhancing availability, but rather in its use of these system properties to enable other techniques such as realistic online self-testing and operator training.

Online verification has appeared in previous systems in the form of redundant data with checking and repair logic, *e.g.*, ECC, and as multi-way component replication with comparison and voting logic. Online self-testing in the form of "built-in self-test" (BIST) techniques has appeared in circuits and embedded systems [32], but not in server or cluster systems. IBM mainframes such as the 3090 and ES/9000 incorporate the ability to inject faults into hardware data and control paths and to invoke recovery code, but these mechanisms have been used for offline testing rather than for revealing latent errors online [24].

Support for problem diagnosis has been studied in the context of root cause analysis. Banga describes a combination of monitoring, protocol augmentation, and cross-layer data correlation techniques to identify the causes of network problems in Network Appliance servers [1]. Other researchers have studied event correlation techniques to isolate potential root causes from a component dependency graph and observed symptoms and alarms [9] [16] [20] [36]. Our work builds on these techniques by proposing more extensive hardware and software logging, by potentially offloading diagnostic work onto a separate diagnostic processor, and by dynamically building dependency information by tracing system requests end-to-end as they pass through the system.

The importance of human operators in maintaining system availability has been recognized for decades [12] [28], but research into reducing human error in server systems through improved human-computer interaction techniques has been all but ignored and its potential impact on availability has not been quantified.

Finally, we note that servers optimized for high density and low power consumption have recently entered the marketplace [30].

6 Conclusion

In this paper we have described the design of ROC-1, a 64-node prototype Internet server based on combined CPU-disk *bricks* that incorporate hardware support for Recovery Oriented Computing (ROC). Emphasizing redundancy and isolation, online testing and verification, support for problem diagnosis, and design for human interaction, ROC recognizes that failures are inevitable and therefore emphasizes reducing the time needed to detect and recover from failures rather than eliminating them. By building the ROC principles into ROC-1's hardware and software from the ground up, we believe we will achieve a significantly higher resilience to hardware, software, and human errors than is achieved using existing techniques. ROC-1's hardware is also built with a concern for cost performance; in an era of Internet servers with largely "embarrassingly parallel" workloads, we believe small, low-power-consuming nodes optimized for density yield better cost-performance than large, power-hungry nodes optimized for SPEC benchmarks.

References

[1] G. Banga. Auto-diagnosis of Field Problems in an Appliance Operating System. *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.

[2] A. Brown. Accepting failure: availability through repair-centric system design. U.C. Berkeley Qualifying Exam Proposal, 2001.

[3] Brown, A. and D. A. Patterson. Embracing Failure: A Case for Recovery-Oriented Computing (ROC). To appear in *Proceedings of the 2001 High Performance Transaction Processing Symposium (HPTS '01)*, 2001.

[4] A. Brown and D. A. Patterson. To Err is Human. *First Workshop on Evaluating and Architecting System dependabilitY (EASY '01)*, 2001.

[5] A. Brown and D.A. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.

[6] G. Candea and A. Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, 2001.

[7] J.D. Case, M. Fedor, M.L. Schoffstall, and C. Davin. *Simple Network Management Protocol* (SNMP), RFC 1157, 1990.

[8] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High performance, reliable secondary storage. *ACM Computing Surveys*, vol. 26, no. 2, pp. 145-185, 1994.

[9] J. Choi, M. Choi, and S. Lee. An Alarm Correlation and Fault Identification Scheme Based on OSI Managed Object Classes. *1999 IEEE International Conference on Communications*, 1999, pp. 1547–51.

[10] Distributed Management Task Force, Inc. *Web-Based Enterprise Management* (WBEM) Initiative, 2001. http://www.dmtf.org/standards/standard_wbem.php

[11] A. Fox, S. Gribble, Y. Chawathe, *et al.* Cluster-based Scalable Network Services. *Proceedings of the 16th Symposium on Operating System Principles (SOSP-16)*, 1997.

[12] J. Goldberg. New Problems in Fault-Tolerant Computing. *Proceedings of the 1975 International Symposium on Fault-Tolerant Computing*, 1975, pp. 29-34.

[13] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. *Proceedings of the 17th Symposium on Operating Systems Principles*, 1999.

[14] J. Gray. Why Do Computers Stop and What Can Be Done About It? *Symposium on Reliability in Distributed Software and Database Systems*, 1986, pp., 3-12.

[15] S. Gribble, M. Welsh, R. von Behren, *et al.* The Ninja architecture for robust Internet-scale systems and services. *Computer Networks* 35(4):473-497.

[16] B. Gruschke. Integrated Event Management: Event Correlation Using Dependency Graphs. *Proceedings of 9th IFIP/IEEE International Workshop on Distributed Systems Operation & Management (DSOM98)*, 1998.

[17] J. Hamilton. Fault Avoidance vs. Fault Tolerance: Testing Doesn’t Scale. *High Performance Transaction Systems (HPTS) Workshop*, 1999.

[18] Y. Huang, C. Kintala, N. Kolettis et al. Software Rejuvenation: Analysis, Module and Applications. *Proceedings of the 1995 International Symposium on Fault-Tolerant Computing*, 1995, pp. 381–390.

[19] ISO/DIS 11898. Controller Area Network (CAN) for High Speed Communication. 1992.

[20] S. Kätker and M. Paterok. Fault Isolation and Event Correlation for Integrated Fault Management. *Fifth IFIP/IEEE International Symposium on Integrated Network Management (IM V)*, 1997, pp. 583–596.

[21] R. Kembel. *The Fibre Channel Consultant: A Comprehensive Introduction.* Northwest Learning Assoc., 1998.

[22] D. R. Kuhn. Sources of Failure in the Public Switched Telephone Network. *IEEE Computer* 30(4), April 1997.

[23] J. Menn. Prevention of Online Crashes is No Easy Fix. *Los Angeles Times*, 2 December 1999, C-1.

[24] A. C. Merenda and E. Merenda. Recovery/Serviceability System Test Improvements for the IBM ES/9000 520 Based Models. *Proceedings of the 1992 International Symposium on Fault-Tolerant Computing*, 1992, pp. 463–467.

[25] B. Murphy and T. Gent. Measuring System and Software Reliability using an Automated Data Collection Process. *Quality and Reliability Engineering International*, 11:341–353, 1995.

[26] Ninja: A Framework for Network Services. *Submission to the 18th Symposium on Operating System Principles (SOSP)*, 2001.

[27] C. Perrow. *Normal Accidents*. Princeton University Press, 1999.

[28] J. Rasmussen and W. Rouse, eds. *Human Detection and Diagnosis of System Failures: Proceedings of the NATO Symposium on Human Detection and Diagnosis of System Failures.* Plenum Press, 1981.

[29] J. Reason. *Human Error.* Cambridge University Press, 1990.

[30] RLX Technologies. “Redefining server economics.” RLX Technologies White Paper, 2001. http://www.rocketlogix.com/

[31] SPEC, Inc. *SPECmail 2001*. http://www.spec.org/osg/mail2001/

[32] A. Steininger and C. Scherrer. On the Necessity of On-line-BIST in Safety-Critical Applications—A Case-Study. *Proceedings of the 1999 International Symposium on Fault-Tolerant Computing*, 1999, pp. 208–215.

[33] Sun Microsystems, Inc. Java Management Extensions JMX. *Preliminary Specification Draft* 1.9, Sun Microsystems, Inc., 1999.

[34] T. Sweeney. No Time for DOWNTIME—IT Managers feel the heat to prevent outages that can cost millions of dollars. *InternetWeek*, n. 807, 3 April 2000.

[35] J. Waddle and M. Walker. Power Dependence Determination with Powerline Networking. Project for UC Berkeley CS252, May 2001. http://www.cs.berkeley.edu/~mwalker/powernet.html

[36] S. Yemini, S. Kliger et al. High Speed and Robust Event Correlation. *IEEE Communications Magazine*, 34(5):82–90, May 1996.