Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies

David Patterson, Aaron Brown, Pete Broadwell, George Candea[†], Mike Chen, James Cutler[†], Patricia Enriquez*, Armando Fox[†], Emre Kıcıman[†], Matthew Merzbacher*, David Oppenheimer, Naveen Sastry, William Tetzlaff[‡], Jonathan Traupman, and Noah Treuhaft

Computer Science Division, University of California at Berkeley (unless noted)
*Computer Science Department, Mills College, Oakland, California
[†]Computer Science Department, Stanford University, Palo Alto, California
[‡]IBM Research, Almaden, California
Contact Author: David A. Patterson, patterson@cs.berkeley.edu

Computer Science Technical Report UCB//CSD-02-1175, U.C. Berkeley
March 15, 2002

Abstract: It is time to broaden our performance-dominated research agenda. A four order of magnitude increase in performance since the first ASPLOS in 1982 means that few outside the CS&E research community believe that speed is the only problem of computer hardware and software. Current systems crash and freeze so frequently that people become violent.[1] Fast but flaky should not be our 21st century legacy.

Recovery Oriented Computing (ROC) takes the perspective that hardware faults, software bugs, and operator errors are facts to be coped with, not problems to be solved. By concentrating on Mean Time to Repair (MTTR) rather than Mean Time to Failure (MTTF), ROC reduces recovery time and thus offers higher availability. Since a large portion of system administration is dealing with failures, ROC may also reduce total cost of ownership. One to two orders of magnitude reduction in cost mean that the purchase price of hardware and software is now a small part of the total cost of ownership.

In addition to giving the motivation and definition of ROC, we introduce failure data for Internet sites that shows that the leading cause of outages is operator error. We also demonstrate five ROC techniques in five case studies, which we hope will influence designers of architectures and operating systems.

If we embrace availability and maintainability, systems of the future may compete on recovery performance rather than just SPEC performance, and on total cost of ownership rather than just system price. Such a change may restore our pride in the architectures and operating systems we craft.

1. Motivation

The focus of researchers and developers for the 20 years since the first ASPLOS conference has been performance, and that single-minded effort has yielded a 12,000-fold improvement [HP02]. Key to this success has been benchmarks, which measure progress and reward the winners.

Not surprisingly, this single-minded focus on performance has neglected other aspects of computing: dependability, security, privacy, and total cost of ownership (TCO), to name a few. For example, TCO is widely reported to be 5 to 10 times the purchase price of hardware and software, a sign of neglect by our community. We were able to reverse engineer a more detailed comparison from a recent survey on TCO for cluster-based services [Gillen02]. Figure 1 shows that the TCO/purchase ratios we found are 3.6 to 18.5. The survey suggests that a third to half of TCO is recovering from or preparing against failures.

Such results are easy to explain in retrospect. Several trends have lowered the purchase price of hardware and software: Moore's Law, commodity PC hardware, clusters, and open source software. Indeed, the ratio is higher in Figure 1 for clusters using open source and PC hardware. In contrast, system administrator salaries have increased while prices have dropped. Moreover, faster processors and bigger disks mean more users on these systems, and it is likely that system administration cost is more a function

[1] A Mori survey in Britain found that more than 12% have seen their colleagues bully the IT department when things go wrong, while 25% of under 25 year olds have seen peers kicking their computers. Some 2% claimed to have actually hit the person next to them in their frustration. HCI Prof. Helen Petrie says, "… it starts in the mind, then becomes physical, with shaking, eyes dilating, sweating, and increased heart rate. You are preparing to have a fight, with no one to fight against." From *Net effect of computer rage*, by Mark Hughes-Morgan, AP, 2/25/02.

of the number of users than of the price of the system. These trends inevitably lead to purchase price of hardware and software becoming a dwindling fraction of the total cost of ownership.

Our concentration on performance may have led us to neglect availability. Despite marketing campaigns promising 99.999% availability, well-managed servers today achieve 99.9% to 99%, or 8 to 80 hours of downtime per year. Each hour can be costly, from $200,000 per hour for an Internet service like Amazon to $6,000,000 per hour for a stock brokerage firm [Kembe00].

Operating system/Service	Linux/Internet	Linux/Collab.	Unix/Internet	Unix/Collab.
Average number of servers	3.1	4.1	12.2	11.0
Average number of users	1150	4550	7600	4800
HW-SW purchase price	$127,650	$159,530	$2,605,771	$1,109,262
3 year Total Cost of Ownership	$1,020,050	$2,949,026	$9,450,668	$17,426,458
TCO/HW-SW ratio	8.0	18.5	3.6	15.7

Figure 1. Ratio of three tear total cost of ownership to hardware-software purchase price. TCO includes administration, operations, network management, database management, and user support. Several costs typically associated with TCO were *not* included: space, power, backup media, communications, HW/SW support contracts, and downtime. The sites were divided into two services: "Internet/Intranet" (firewall, Web serving, Web caching, B2B, B2C) and "Collaborative" (calendar, email, shared files, shared database). IDC interviewed 142 companies, with average sales of $2.4B/year, to collect these statistics.

We conducted two surveys on the causes of downtime, with unexpected results. In our first survey, we collected failure data on the U.S. Public Switched Telephone Network (PSTN). In our second, we collected failure data from three Internet sites. Based on that data, Figure 2 shows the percentage of failures due to operators, hardware failures, software failures, and overload. The surveys are notably consistent in their suggestion that operators are the leading cause of failure.

We are not alone in calling for new challenges. Jim Gray [1999] has called for *Trouble-Free Systems*, which can largely manage themselves while providing a service for millions of people. Butler Lampson [1999] has called for systems that *work*: they meet their specs, are always available, adapt to changing environment, evolve while they run, and grow without practical limit. Hennessy [1999] has proposed a new research target: availability, maintainability, and scalability. IBM Research [2001] has announced a new program in Autonomic Computing, whereby they try to make systems smarter about managing themselves rather than just faster. Finally, Bill Gates [2002] has set *trustworthy systems* as the new target for his developers, which means improved security, availability, and privacy.

The Recovery Oriented Computing (ROC) project presents one perspective on how to achieve the goals of these luminaries. Our target is services over the network, including both Internet services like Yahoo! and enterprise services like corporate email. The killer metrics for such services are availability and total cost of ownership, with Internet services also challenged by rapid scale-up in demand and deployment and rapid change in software.

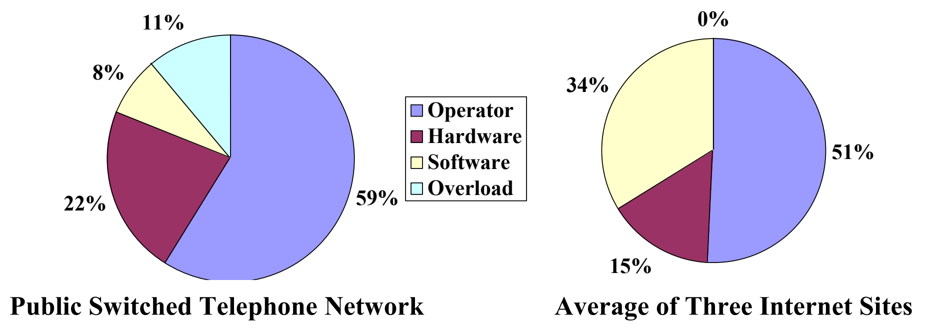


Figure 2. Percentage of failures by operator, hardware, software, and overload for PSTN and three Internet sites. Note that the mature software of the PSTN is much less of a problem than Internet site software, yet the Internet sites have such frequent fluctuations that they have overprovisioned so that overload failures are rare. The PSTN data measured blocked calls in the year 2000. We collected this data from the FCC; it represents over 200 telephone outages in the U.S. that affected at least 30,000 customers or lasted at least 30 minutes. Rather than report outages, telephone switches record the number of attempted calls blocked during an outage, which is an attractive metric. (This figure does not show vandalism, which is responsible for 0.5% of blocked calls.) The Internet site data measured outages in 2001. We collected this data from companies in return for anonymity; it represents six weeks to six months of service for 500 to 5000 computers. (The figure does not include environmental causes, which are responsible for 1% of the outages. Also, 25% of outages had no identifiable cause and are not included in the data.)

Section 2 of this paper surveys other fields, from disaster analysis to civil engineering, to look for new ideas for dependable systems. Section 3 presents the ROC hypotheses of concentrating on recovery to make systems more dependable and less expensive to own, and lists several ROC techniques. The next five sections each evaluate one ROC technique in the context of a case study. Given the scope of the ROC hypotheses, our goal in this paper is to provide enough detail to demonstrate that the techniques are plausible. Section 9 contains 80 references to related work, indicating the wide scope of the ROC project. Section 10 concludes with a discussion and future directions for ROC.

The authors hope that architects and OS developers will consider their plans from a ROC perspective.

2. Inspiration From Other Fields

Since current systems are fast but failure prone, we decided to look to other fields for new directions and ideas. We surveyed three fields: disaster analysis, human error analysis, and civil engineering design.

2.1 Disasters and Latent Errors in Emergency Systems

Charles Perrow [1990] analyzed disasters, such as the one at the nuclear reactor on Three Mile Island (TMI) in Pennsylvania in 1979. To try to prevent disasters, nuclear reactors are redundant and rely heavily on "defense in depth," meaning multiple layers of redundant systems.

Reactors are large, complex, tightly coupled systems with lots of interactions, so it is hard for operators to understand the state of the system, its behavior, or the potential impact of their actions. There are also errors in implementation and in the measurement and warning systems which exacerbate the situation. Perrow points out that in tightly coupled complex systems bad things will happen, which he calls *normal accidents*. He says seemingly impossible multiple failures–which computer scientists normally disregard as statistically impossible–do happen. To some extent, these are correlated errors, but latent errors also accumulate in a system, awaiting a triggering event.

He also points out that emergency systems are often flawed. Since unneeded for day-to-day operation, only an emergency tests them, and latent errors in emergency systems can render them useless. At TMI, two emergency feedwater systems had shutoff valves in the same location, and both were set to the wrong position. When the emergency occurred, these redundant backup systems failed. Ultimately, the containment building itself was the last defense, and they finally did get enough water to cool the reactor. However, in breaching several levels of defense, the core was destroyed.

Perrow says operators are blamed for disasters 60% to 80% of the time, and TMI was no exception. However, he believes that this number is much too high. People who designed the system typically do the postmortem, where hindsight determines what the operators should have done. He believes that most of the problems are in the design itself. Since there are limits to how many errors can be eliminated through design, there must be other means to mitigate the effects when "normal accidents" occur.

Our lessons from TMI are the importance of removing latent errors, the need for testing recovery systems to ensure that they will work when needed, the need to help operators cope with complexity, and the value of multiple levels of defense.

2.2 Human Error and Automation Irony

Because of TMI, researchers began to look at why humans make errors. James Reason [1990] surveys the literature of that field and makes some interesting points. First, there are two kinds of human error. *Slips* or *lapses*–errors in execution–where people do not do what they intended to do, and *mistakes*–errors in planning–where people do what they intended to do, but chose the wrong course. Second, training can be characterized as creating mental production rules to solve problems, and normally what we do is rapidly go through production rules until we find a plausible match. Thus, humans are furious pattern matchers. Third, we are poor at solving problems from first principles, and can only do so for only so long before our brains “tire.” Cognitive strain leads us to try least-effort solutions first, typically from our production rules, even when wrong. Fourth, humans self-detect errors. According to Reason, people detect about 75% of errors immediately after they make them. He concludes that human errors are inevitable.

A major observation from the field of human error research, labeled *the Automation Irony*, is that automation does not cure human error. The reasoning is that once designers realize that humans make errors, they often try to design a system that reduces human intervention. Automation usually addresses the easy tasks for humans, leaving to the operator the complex, rare tasks that were not successfully automated. Humans, who are not good at solving problems from first principles, are ill suited to such tasks, especially under stress. The irony is that automation reduces the chance for operators to get hands-on control experience, preventing them from building mental production rules and models for troubleshooting. Thus, automation often decreases system visibility, increases system complexity, and limits opportunities for interaction, all of which can make it harder for operators to use and make it more likely for them to make mistakes.

Our lessons from human error research are that human operators will always be involved with systems and that humans will make errors, even when they truly know what to do. The challenge is to design systems that are synergistic with human operators, ideally giving operators a chance to familiarize themselves with systems in a safe environment, and to correct their own errors.

2.3 Civil Engineering and Margin of Safety

Perhaps no engineering field has embraced safety as much as civil engineering. Petroski [1992] has said that this was not always the case. With the arrival of the railroad in the 19th century, engineers had to learn how to build bridges that could support fast-moving vehicles that weighed tons.

They were not immediately successful: between the 1850s and 1890s about a quarter of all iron truss railroad bridges failed! To correct that situation, engineers started studying failures, as they learned more from bridges that fell than from those that didn't. Second, they started to add redundancy so that some pieces could fail yet bridges would survive. However, the major breakthrough was the concept of a *margin of safety*; engineers would enhance their designs by a factor of 3 to 6 to accommodate the unknown. The safety margin compensates for flaws in building material, construction mistakes, overloading, and even design errors. Since humans design, build, and use bridges, and since human errors are inevitable, the margin of safety is necessary. Also called the *margin of ignorance*, it allows safe structures without having to know everything about the design, implementation, and future use of a structure. Despite use of supercomputers and mechanical CAD to design bridges in 2002, civil engineers still multiply the calculated load by a small integer to be safe.

A cautionary tale on the last principle comes from RAID. Early RAID researchers were asked what would happen to RAID-5 if it used a bad batch of disks. Their research suggested that as long as there were standby spares on which to rebuild lost data, RAID-5 would handle bad batches, and so they assured others. A system administrator told us recently that every administrator he knew had lost data on RAID-5 at one time in his career, although they had standby spare disks. How could that be? In retrospect, the quoted MTTF of disks assume nominal temperature and limited vibration. Surely, some RAID systems were exposed to higher temperatures and more vibration than anticipated, and hence had failures much more closely correlated than predicted. A second problem that sometimes occurs in RAID systems is operator removal of a good disk instead of a failed disk, thereby inducing a second failure. Whether this is a slip or a mistake, data is lost. Had our field embraced the principle of the margin of safety, the RAID papers would have said that RAID-5 was sufficient for faults we could anticipate, but recommend RAID-6 (up to two disk failures) to accommodate the unanticipated faults. If so, there might have been significantly fewer data outages in RAID systems.

Our lesson from civil engineering is that the justification for the margin of safety is as applicable to servers as it is to structures, and so we need to understand what a margin of safety means for our field.

3. ROC Hypotheses: Repair Fast to Improve Dependability and to Lower Cost of Ownership

"If a problem has no solution, it may not be a problem,
but a fact, not to be solved, but to be coped with over time." –Shimon Peres

The Peres quote above is the guiding proverb of Recovery Oriented Computing (ROC). We consider errors by people, software, and hardware to be facts, not problems that we must solve, and fast recovery is how we cope with these inevitable errors. Since unavailability is approximately MTTR/MTTF, shrinking time to recover by a factor of ten is just as valuable as stretching time to fail by a factor of ten. From a research perspective, we believe that MTTF has received much more attention than MTTR, and hence there may be more opportunities for improving MTTR. The first ROC hypothesis is that *recovery performance* is more fruitful for the research community and more important for society than traditional performance in the 21st century. Stated alternatively, Peres' Law will soon be more important than Moore's Law.

A side benefit of reducing recovery time is its impact on cost of ownership. Lowering MTTR reduces money lost to downtime. Note that the cost of downtime is not linear. Five seconds of downtime probably costs nothing, five hours may waste a day of wages and a day of income of a company, and five weeks may drive a company out of business. Thus, reducing MTTR may have nonlinear benefits on cost of downtime (see Section 4). A second benefit is reduced cost of administration. Since a third to half of a system administrator's time may be spent recovering from failures or preparing for the possibility of failure before an upgrade, ROC may also lower the human component of cost of ownership. The second ROC hypothesis is that research opportunities and customer emphasis in the 21st century will be on total cost of ownership rather than on the conventional measure of purchase price of hardware and software.

Although we are more interested in the research opportunities of MTTR, we note that our thrust complements research in improving MTTF, and we welcome it. Given the statistics in section 2, there is no danger of hardware, software, and operators becoming perfect: MTTR will remain relevant. Also, ROC techniques record failures, which can be used to improve software and thereby stretch its MTTF.

Progress on performance was so quick in part because we had a common yardstick–benchmarks–to measure success. To make such rapid progress on recovery, we need similar incentives. Prior work has successfully benchmarked availability [Brown00][Brown02], and so we do not cover the topic in this paper. With any benchmark, one of the first questions is whether it is realistic. Rather than guess why systems fail, we need facts on which to base fault workloads. Section 2 presents the type of data that we need to drive the benchmarks.

Although tales from disasters and outages seem daunting, the ROC hypotheses are applied to a virtual world, allowing us to do things that are impossible in the physical world. This view may simplify our task. For example, civil engineers must design walls to survive a large earthquake, but in a virtual world it may be just as effective to let it fall and then replace it milliseconds later, or to have several walls standing by in case one fails.

Our search for inspiration from other fields led to new techniques as well as some commonly used. Six techniques that guide ROC are: 1) Recovery experiments to test repair mechanisms in development and in the field; 2) Aids for diagnosing the causes of errors in live systems; 3) Partitioning to rapidly recover from faults and to contain them; 4) Reversible systems to handle undo and provide a safety margin; 5) Defense in depth in case the first line of defense does not contain an error; and 6) Redundancy to survive faults and failing fast to reduce MTTR. This list is meant to be suggestive rather than exhaustive.

As this is a conference-length paper, we had the choice of long sections on one or two topics or giving shorter versions of all techniques to demonstrate the viability of the whole ROC vision. We chose the latter approach, referencing technical reports if readers wish to learn more. The next five sections are case studies in which we tried the first five techniques. The sixth technique is already well established and widely used.

4. Software Recovery Experiments: FIG

We do not expect advances in recovery until it is as easy to test recovery as it is to test functionality and performance. Recovery experiments are needed not only in the development lab, but also in the field to show us what happens when a fault occurs in a given system with its unique combination of hardware, software, and firmware. Although others have tested software interfaces via random inputs (see Section 9), the software infrastructure is usually neither portable nor easy to use. This case study shows the value of recovery experiments based on an easy-to-use, low-overhead interposition tool that finds unusual behavior even in mature software.

FIG (Fault Injection in glibc) is a lightweight, extensible tool for triggering and logging errors at the application/system boundary. FIG runs on UNIX-like operating systems, using the LD_PRELOAD environment variable to interpose itself between the application and glibc, the GNU C library. When FIG intercepts a call, it then chooses, based on directives from a control file, whether to allow the call to complete normally or to return an error that simulates a failure in the operating environment. We measured this interposition overhead at 5%. Although FIG targets functions in glibc, it can be adapted easily to other libraries.

To test the effectiveness of FIG, we started with five mature applications: the text editor Emacs (with and without the X Window System), the browser Netscape; the database library Berkeley DB (with and without transactions); the database MySQL, and the HTTP server Apache. We reasoned that if recovery experiments can help evaluate the dependability of mature software, then they will surely benefit new software.

We triggered errors in a dozen system calls, and Figure 4 shows results for `read()`, `write()`, `select()`, and `malloc()`. Emacs fared much better without X: EIO and ENOMEM errors caused crashes with X, and more acceptable behavior without it. Netscape exited cleanly but gave no user warning at the first EIO or ENOSPC, and aborted page loads on EINTR. Overall, it had the most questionable behavior. As expected, Berkeley DB in non-transactional mode did not handle errors gracefully, as write errors could corrupt the database. In transactional mode, it detected and recovered properly from all but memory errors. MySQL and Apache were the most robust.

System Call	`read()`		`write()`		`select()`	`malloc()`
Error	EINTR	EIO	ENOSPC	EIO	ENOMEM	ENOMEM
Emacs - no X Window	O.K.	exits	warns	warns	O.K.	crash
Emacs - X Window	O.K.	crash	O.K.	crash	crash / exit	crash
Netscape	warn	exit	exit	exit	n/a	exit
Berkeley DB - Xact	retry	detected	xact abort	xact abort	n/a	xact abort, crash
Berkeley DB - no Xact	retry	detected	data loss	data loss	n/a	detected, data loss
MySQL	xact abort	retry, warn	xact abort	xact abort	retry	restart
Apache	O.K.	request dropped	request dropped	request dropped	O.K.	n/a

Figure 4. Reaction of applications to faults triggered in four system calls. EINTER = Interrupted system call, EIO = general I/O error, ENOSPC = insufficient disk space, ENOMEM = insufficient memory. On seeing ENOMEM for malloc(), Berkeley DB would occasionally lose data or crash. Write errors lost data at high, correlated error rates.

One lesson from FIG is that even mature, reliable programs have misdocumented interfaces and poor error recovery mechanisms. We conclude that application development can benefit from recovery experiments such as those conducted here with FIG. One question is whether the faults are typical. As part of our failure collection efforts (see Section 2), we plan to characterize what types of library errors occur during failures.

Even with this limited number of examples, FIG allows us to see both successful and unsuccessful approaches to dependable application programming. Four examples of successful practices are:

- *Resource Preallocation*. Apache was one application which did not crash with an error in `malloc()`. On our workload, Apache allocated all necessary memory at startup, entirely avoiding `malloc()` failures in the middle of its processing. Resource preallocation does not fully solve the problem, however, since resources may be scarce at program initialization, causing an error. Encountering a failure at program initialization seems more desirable than doing so in the middle, when the system will is more likely to be disrupted by abnormal termination of an operation.
- *Graceful Degradation.* Techniques that offer partial service in the face of failures allow the service to degrade gracefully. Such techniques, whereby errors lead to reduced functionality rather than outright failures, postpone downtime until an operator can fully recover the system. Apache again provides an example: when its log file cannot be written, the rest of the service continues without interruption, but without logging.
- *Selective Retry*. Retrying errors is a natural solution to resource exhaustion. Waiting and retrying a failed system call can help if resources later become available. Although we did not include the *ls* program in Figure 4, it uses this technique to retry `malloc()` errors. Instead of retrying indefinitely, it maintains a global count of the number of `malloc()` failures encountered and terminates when the count exceeds a threshold.
- *Process pools*. Once the MySQL server is up and running, it provides a high degree of availability by using a pool of child processes to process queries. As we observed with `malloc()`failures, a child process simply restarts when it encounters an error.

5. Automatic Diagnosis: Pinpoint

Clearly, a failing system should help the operator determine what to fix, as this will reduce MTTR. In the fast moving environment of Internet services, the challenge is to provide aids that can track rapid software change. Pinpoint is a diagnostic tool that tracks changes in modular software. The key technique is the use of standard data mining programs to search traces of successful and unsuccessful user requests; this offers accurate diagnoses with a low rate of false positives. Logging of such errors can also help with repair once the failing module is identified.

Classical error determination starts by making an accurate model of the system. Symptoms are then recorded and a variety of statistical techniques are used to identify a suspect module. The main problem of this *dependency model* approach is that Internet service software changes weekly or even daily, so an accurate model is neither cheap nor likely. A second problem is that the model typically captures only the *logical* structure of a system. Large Internet services have thousands of redundant computers and processes, so we need to know which *instance* of a module is the problem, not just which type of module.

The good news is that many Internet and enterprise services use middleware to construct their systems, such as Microsoft .NET or Java 2 Platform Enterprise Edition (J2EE). Both seek to simplify applications by basing them on standardized, modular components, by providing a set of useful services to those components, and by handling many details of application behavior automatically.

We tried a different path to error determination in J2EE applications, named *Pinpoint*. First, we added monitoring to the J2EE middleware to trace each request to see which instances of modules it uses. Next, we logged each trace and recorded the success or failure of each user request in the trace. Finally, we used standard data mining techniques on the traces to identify faulty modules. By analyzing the components used in the failed requests, but not used in successful requests, we find the culprits. This analysis detects individual faulty components as well as faults occurring due to interactions among multiple components.

Rather than requiring us to build and maintain a system model, Pinpoint automatically tracks changes in the software system, tracing and reporting problems with instances rather than with the logical modules. The only application-specific component required by Pinpoint is checks to determine the success or failure of requests; no application component needs to be modified. Hence, Pinpoint can aid problem determination for almost any J2EE application.

The questions to be answered are: 1) What is the overhead of tracing? 2) How often does it find the problem modules (*hit rate*)? 3) How often does it supply the wrong modules (*false positive rate*)? 4) If the hit rate and false positive rate vary, how do we compare the two approaches?

To answer the first question, we compared the performance of an application hosted on an unmodified J2EE server and one hosted on a Pinpoint server. The overhead was 8%. Given the advantages of not creating and maintaining a model, we believe this is a reasonable trade-off.

Figure 5 answers the next three questions by plotting the hit rate on the on the Y-axis against the false positive rate on the X-axis. As we change parameters for Pinpoint and the simple dependency model, we can see the change in hit rate and false positive rate. Generally, changing the parameters to increase the chances of identifying faulty modules also increases the chances of nominating healthy ones: that is, the hit rates and false positive rates tend to increase hand-in-hand. Note that Pinpoint consistently has a higher hit rate and a lower false positive rate than traditional dependency analysis.

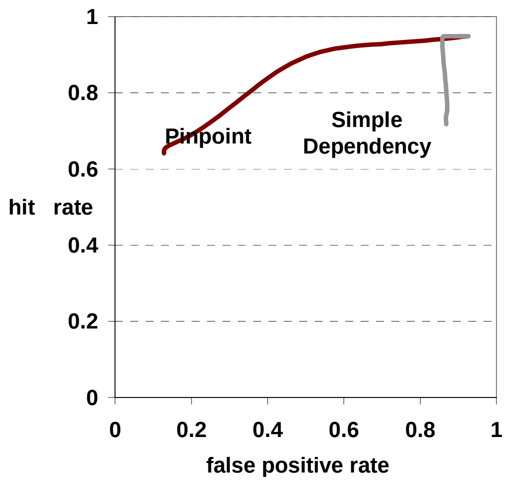


Figure 5. A ROC analysis of hit rate vs. false positive rate. Good results are up (high hit rate) and to the *left* (low false positive rate). For example, if there were 10 real faulty modules, a hit rate of 0.9 with a false positive rate of 0.2 would mean that the system correctly identified 9 of the 10 real ones and erroneously suggested 2 false ones. A sensitivity "knob" can be adjusted; it typically increases both hit rate and false positives. Even at a low sensitivity setting, Pinpoint starts at a false positive rate of just 0.1 combined with a hit rate above 0.6. Raising the sensitivity pushes the false positive rate to 0.5 with a hit rate to 0.9. In contrast, the simple dependency approach starts with a very high false positive rate: 0.9. The sensitivity setting can improve the hit rate from 0.7 to 0.9, but the false positive rate starts at 0.9 and stays there. To validate our approach, we ran the J2EE Pet Store demonstration application and systematically triggered faults in the system over a series of runs. We ran 55 tests that included single-component faults and faults triggered by component interactions. The graph shows single component faults. It is important to note that our fault injection system is separate from our fault detection system. The data mining method used for the Pinpoint analysis was the unweighted pair-group method using arithmetic averages with the Jaccard similarity coefficient. [Jain 1988] For more detail, see [Chen02].

We were pleased with both the accuracy and overhead of Pinpoint, especially given its ability to match the dynamic and replicated nature of Internet services. One limitation of Pinpoint is that it cannot distinguish between sets of tightly coupled components that are always used together. We are looking at inserting test inputs to isolate such modules. Another limitation of Pinpoint, as well as existing error determination approaches, is that it does not work with faults that corrupt state and affect subsequent requests. This interdependency makes it difficult to detect the real faults because the subsequent requests may fail while using a different set of components.

6. Fine Grained Partitioning and Recursive Restartability

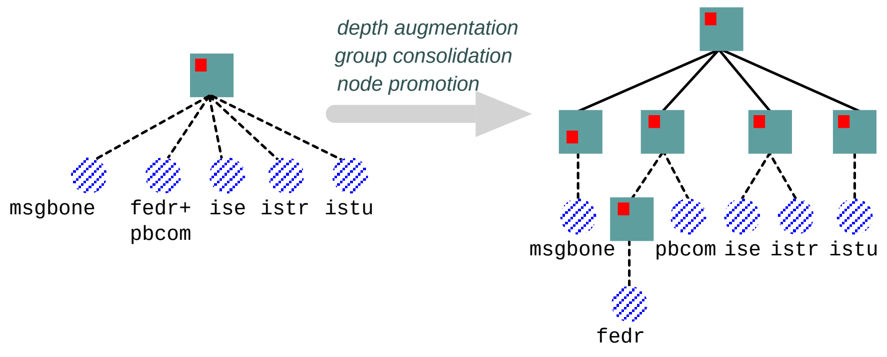
Once the operator knows what to fix, we can partition hardware to isolate failures or software to help reduce the time to restart. This case study concerns *Mercury*, a satellite ground station in which we employed fine-grained partial restarts to reduce the MTTR of the control software by a factor of almost six. Besides being a significant quantitative improvement, it also constituted a qualitative improvement that lead to nearly 100% availability of the ground station during the critical period when the satellite passes overhead.

Recursive restartability [Candea01a] is an approach to system recovery that assumes that, in critical infrastructures, most bugs cause software to crash, deadlock, spin, leak memory, or otherwise fail in a way that leaves reboot or restart as the only means of restoring the system [Brewer01][Gray78]. Reboots are an effective and efficient workaround because they are easy to understand and to employ, they reliably reclaim resources, and they unequivocally return software to its start state, which is generally the best tested and understood state of the system.

Unfortunately, most systems do not tolerate unannounced restarts, resulting in long downtimes and potential data loss. A recursively restartable (RR) system, however, gracefully tolerates successive restarts at multiple levels. Fine-grained partitioning enables bounded, partial restarts that recover a failed system faster than a full reboot. RR also enables strong fault containment [Candea01b].

We applied the RR ideas to Mercury, a ground station that controls communication with two orbiting satellites. Mercury is built from off-the-shelf antennas and radios, driven by x86-based PCs running Linux, with most of the software written in Java. Mercury breaks with the satellite community tradition by emphasizing low production cost over mission criticality.

Figure 6 shows a recursively restartable system described by a restart tree–a hierarchy of restartable components in which nodes are highly failure-isolated. A restart at any node results in the restart of the entire subtree rooted at that node. A restart tree does not capture the *functional* dependencies between



Failed component	msgbone	fedr	pbcom	ise/istr	istu
MTTF	1 month	10 min	1 month	5 hours	5 hours
MTTR before	**28.9** sec	**28.9** sec	**28.9** sec	**28.9** sec	**28.9** sec
MTTR after	**4.7** sec	**5.0** sec	**21.9** sec	**6.1** sec	**5.8** sec

Figure 6. Applying Recursive Restartability to a satellite ground station. The Mercury instances before and after applying RR are described by the two restart trees. Mercury consists of a number of components: a communication backbone for XML messages (msgbone), an XML command translator (fedr), a serial port to TCP socket mapper (pbcom), a position and azimuth estimator (ise), a satellite tracker (istr), and a radio tuner (istu). The first row of the table indicates the observed MTTFs of these components based on 2 years of operation. The last two rows show the effect of the transformations on the system's time-to-recover as a function of the component that fails.

components, but rather the *restart* dependencies. Subtrees represent *restart groups*, which group together components with common restart requirements. Based on information of which component failed (for example, using Pinpoint (see Section 5), an *oracle* dictates which node in the tree must be restarted. The goal of a perfect oracle is to always choose a minimal set of subsystems that need restarting, in order to minimize the time-to-recover. If a particular restart does not cure the failure, the oracle may choose to "hike" the tree and restart at a higher level. Restarting groups that are higher up in the restart tree leads to a longer but higher-confidence recovery.

To minimize time-to-recover in Mercury, we applied a number of transformations to balance the restart tree, looking at the frequency of occurrence and the severity of the restart time [Candea02]. The key insight is that balancing the tree by looking at MTTR produces much better results than balancing based on logical structure. Figure 6 summarizes the result of these transformations. The most compelling result is a six-fold reduction in recovery time for failures occurring in `fedr`, which we achieved by splitting it into a part that failed often (due to JVM crashes) but recovered quickly, and a part that failed infrequently but recovered slowly. Using the frequency of failures in Figure 6, the overall weighted average recovery time for Mercury fell from 28.9 seconds to 5.6 seconds due to RR.

Downtime under a heavy or critical workload is more expensive than downtime under a light or non-critical workload, and unplanned downtime is more expensive than planned downtime [Brewer01]. Mercury constitutes an interesting example in the mission-critical service space: downtime during satellite passes is very expensive because we may lose science data and telemetry. Worse, if the failure involves the tracking subsystem and the recovery time is too long, we may lose the whole pass: the antenna and spacecraft can get sufficiently misaligned that the antenna cannot catch up when the tracking subsystem comes back online. A large MTTF does not guarantee a failure-free pass, but a short MTTR can help guarantee that we will not lose the whole pass because of a failure. As mentioned in Section 3, this is another example where the cost of downtime is not linear in its length. Applying recursive restartability to a mission-critical system allowed us to reduce its time-to-recover from various types of failures and achieve nearly 100% availability with respect to satellite passes.

7. Reversible Systems for Operators: Undoable Email System

Although partitioning and recursive restartability can help with software and hardware faults, we need to recover from operator errors as well. We believe that undo for operators would be a significant advance. To achieve this goal, we must preserve the old data and log the new inputs. Fortunately, some commercial file systems today offer such features, and disk capacity is the fastest growing computer technology. Such a system also provides a margin of safety for large classes of hardware, software, and operator errors, since the operator can restore the system to its pre-fault state. It also makes it safer to trigger faults in a live system to test how they react in an emergency, since there is a fallback in case the partitioning fences fail.

This case study concerns an Undo layer designed to provide a forgiving environment for human system operators. We have two goals here. The first is to provide a mechanism that allows operators to recover from their inevitable mistakes; recall from Section 2 that operator errors are a leading cause for service failures today. The second goal is to give operators a tool that allows them to retroactively repair latent errors that went undetected until too late. Here we leverage the fact that computers operate in a virtual world in which "time travel" is possible: by rolling back virtual state, the effects of latent errors can often be reversed. This is unlike physical systems like Three Mile Island.

We have chosen email as the target application for our first implementation of a ROC Undo layer. Email has become an essential service for today's enterprises, often acting as the communications "nervous system" for businesses and individuals alike, and it is one of the most common services offered by network service vendors. Email systems also offer many opportunities for operator error and retroactive repair. Examples of operator error that can be addressed by our undo layer include misconfigured filters (spam, antivirus, procmail, and so on) that inadvertently delete user mail, accidental deletion of user mailboxes, mail corruption when redistributing user mailboxes to balance load, and installation of buggy or broken software upgrades that perform poorly or corrupt mail. Retroactive repair is useful in an email system when viruses or spammers attack: with an undo system, the operator can "undo" the system back to the point before the virus or spam attack began, retroactively install a filter to block the attack, and then "redo" the system back to the present time. Furthermore, the undo abstraction could be presented to the user, allowing the user to recover from inadvertent errors such as accidentally deleting messages or folders without involving a system administrator.

To support retroactive repair and recovery from operator error, our ROC undo model supports a 3-step undo process that we have dubbed "*the three R's*": *Rewind, Repair*, and *Replay*. In the rewind step, all system state (including mailbox contents as well as OS and application hard state) is reverted to its contents at an earlier time (before the error occurred). In the repair step, the operator can make any changes to the system he or she wants. Changes could include fixing a latent error, installing a preventative filter or patch, or simply retrying an operation that was unsuccessful the first time around (like a software upgrade of the email server). Finally, in the replay step, the undo system re-executes all user interactions with the system, reprocessing them with the changes made during the repair step.

There are many challenges in implementing the 3R model. The most important problem is handling *externalized state*, which occurs when the undo cycle alters state already seen by an external user of the system. In email, this problem arises when the undo cycle modifies or deletes messages that a user has already read, forwarded, or replied to; this scenario could occur, for example, if the operator changes incoming mail filters during the repair stage. We handle the resulting inconsistencies by issuing compensating actions. For example, when undo alters an already-read email message, we replace it with a new message that explains to the user why the message was changed or deleted. Similar actions can be used to compensate for mail that should not have been sent, although we don't provide a way to "unsend" email beyond perhaps buffering outgoing messages for a short "undo window". Note that compensation does not conceal the inconsistencies that arise when externalized state is altered; it merely explains the inconsistency and allows the user, typically a human being, to understand why it occurred.

We are building a prototype of our email-targeted undo layer. It operates at the granularity of a single system. It is as a proxy layer that wraps an existing, unmodified IMAP- and SMTP-based mail server. We chose to build the undo layer as a proxy to allow recovery from operator error during major events, such as software upgrades of the mail server. Figure 7 illustrates the operation of the proxy.

Besides the proxy, the other major component of our prototype is a non-overwriting storage layer that sits underneath the mail server. This layer enables the rewind phase of undo by providing access to historical versions of the system's hard state. The time-travel layer can be implemented using file system snapshots (such as those provided by the Network Appliance WAFL file system [Hitz95]), although we are investigating using a more flexible versioning system such as the Elephant file system [Santry99].

Analysis of our departmental mail server for 1270 users indicates that the storage overhead of keeping the undo log is up to 1 GB/day for our unoptimized prototype. Three 120-GB EIDE disks, totaling just $540 in 2002, should store a year of log data. We consider this a modest cost to gain the 3Rs.

8. Defense in Depth: ROC-1

Defense in depth suggests suggest that independent modules can provide backup defense that can improve reliability [Fox00]. Examples are hardware interlocks in Therac [Leveson 1993]; virtual machines for fault

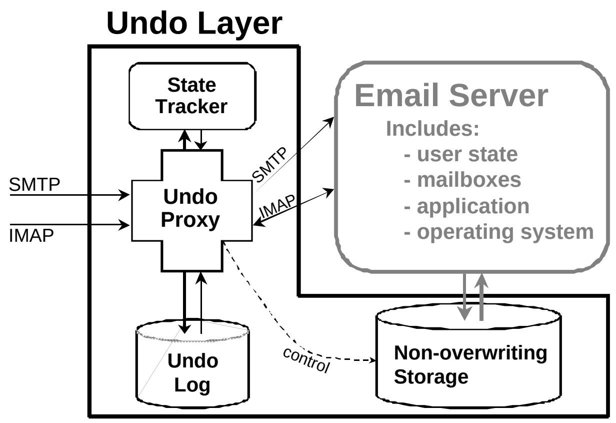


Figure 7. Architecture of undo layer for email. During normal operation, the proxy snoops IMAP and SMTP traffic destined for the mail server and logs mail delivery and user interactions. The proxy also monitors accesses to messages and folders and uses that information to track state that has been made externally visible; this allows the system to issue appropriate compensating actions when the repair/replay cycle invalidates previously-externalized state. Upon an undo request, the non-overwriting storage layer rolls back to the undo point, the operator performs any needed repairs, and then the proxy replays the logged user interactions that were lost during the rollback. The proxy continues to log incoming mail during the undo cycle, although users are only given read-only access until replay has completed. Further discussion of this and other issues raised by the 3R model can be found in [Brown02].

containment and fault tolerance [Bres95]; firewalls from security; and disk and memory scrubbers to repair faults before they accessed by the application.

The ROC-1 hardware prototype is a 64-node cluster composed of custom-built nodes, called *bricks*, each of which is an embedded PC board. Figure 8 shows a brick. For both space and power efficiency, the bricks are each packaged in a single half-height disk canister. Unlike other PCs, each brick has a diagnostic processor (DP) with a separate diagnostic network; the DP provides an independent mechanism to monitor nodes, to isolate failed nodes, and to insert faults. Node isolation and fault insertion are accomplished by removing power from essential chips selectively, including the network interfaces.

ROC-1 does successfully allow the DPs to isolate subsets of nodes; removing power from the network interfaces reliably disconnects nodes from the network. It is less successful at injecting errors by controlling power. Doing so would have required too much board area, and the chips themselves contained too many functions for this power control to be effective.

The lesson from ROC-1 is that we can offer hardware partitioning with standard components via an independent mechanism. However, the high amount of integration per chip suggests that if hardware fault insertion is necessary, chips must change internally and provided interfaces to support such techniques.

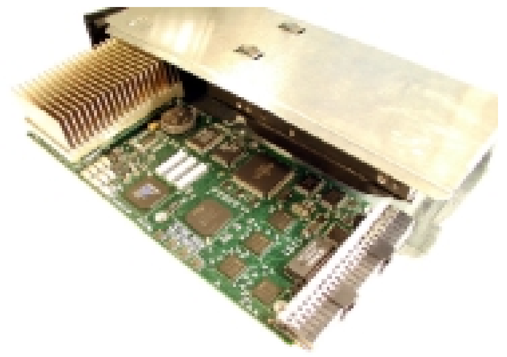


Figure 8. A ROC-1 brick. Each brick contains a 266 MHz mobile Pentium II processor, an 18 GB SCSI disk, 256 MB of ECC DRAM, four redundant 100 Mb/s network interfaces connected to a system-wide Ethernet, and an 18 MHz Motorola MC68376-based diagnostic processor. Eight bricks fit into a tray, and eight trays form the cluster, plus redundant network switches. See [Opp02] for more details.

9. Related Work

The scope of the ROC vision spans many topics; the 80 references mentioned here are a good start.

We are not the first to consider recovery; many research and commercial systems have addressed parts of the ROC agenda presented in this paper. The storage community may come closer than any other to embracing the ROC ideal. Most storage vendors offer products explicitly designed to improve recovery performance; a good example is EMC's TimeFinder system [EMC02], which automatically partitions and replicates storage. EMC suggests that the alternate partitions be used for ROC-like isolated on-line testing and for fast post-crash recovery of large services like Microsoft Exchange. In the research community, recovery-enhancing techniques have emerged serendipitously from work that was originally performance-focused, as in the development of journaling, logging and soft-update file systems [Rosenblum92] [Seltzer93] [Hitz95].

Recovery-oriented work in the OS community is rarer but present. Much of it focuses on the ability to restart quickly after failures. An early example is Sprite's "Recovery Box", in which the OS uses a protected area of non-volatile memory to store crucial state needed for fast recovery [Baker92]. This basic idea of segregating and protecting crucial hard-state to simplify recovery reappears frequently, for example in the derivatives of the Rio system [Chen96][Lowell97][Lowell98], and in recent work on soft-state/hard-state segregation in Internet service architectures [Fox97] [Gribble00]. Most of these systems still use increased performance as a motivation for their techniques; recovery benefits are icing on the cake.

The database community has long paid attention to recovery, using techniques like write-ahead logging [Mohan92] and replication [Gray96] to protect data integrity in the face of failures. Recovery performance has also been a topic of research starting with the POSTGRES system, in which database log format was redesigned to allow near-instantaneous recovery [Stonebraker87]. A more recent example is Oracle 9i, which includes a novel Fast Start mechanism for quick post-crash recovery [Lahiri01] and a limited version of an undo system that allows users to view snapshots of their data from earlier times [Oracle01]. In general, however, transaction performance has far overshadowed recovery performance, probably due to the influence of the performance-oriented TPC benchmarks.

Finally, the traditional fault-tolerance community has occasionally devoted attention to recovery. The best example of this is in the telephone switching infrastructure, which was designed under the assumptions of operator error, buggy software, and faulty hardware, and which coped with these realities through an elaborate array of recovery mechanisms [Meyers77] [AUDITS83]. Unlike the more general-purpose ROC mechanisms, these mechanisms were single-purpose and tailored for the well-defined application of phone switching. Beyond telephone switches, another example of recovery work arising from the fault-tolerance community is the work on software rejuvenation, which periodically restarts system modules to flush out latent errors [Huang95] [Garg97] [Bobbio98]. Another illustration is work on built-in self-test in embedded systems, in which components are designed to proactively scan for possible latent errors and to immediately fail and restart when any are found [Steininger99]. Most of the fault-tolerance community does not believe in Peres's law, however, and therefore focuses on MTTF under the assumption that failures are generally problems that can be known ahead of time and should avoided.

Our ROC approach differentiates itself from this previous work in two fundamental ways. First, ROC treats recovery holistically: a ROC system should be able to recover from any failure at any level of the system, and recovery should encompass all layers. Contrast this with, say, database or storage recovery, where recovery is concerned only with the data in the database/storage and not the entire behavior of the service built on top. Second, ROC covers a much broader failure space than these existing approaches. In particular, ROC addresses human-induced failures, which are almost entirely ignored in existing systems work. ROC also makes no assumptions about what failures might occur. Traditional fault-tolerance work typically limits its coverage to a set of failures predicted by a model; in ROC, we assume that anything can happen and we provide mechanisms to deal with unanticipated failure.

Each ROC technique in this paper draws on a background of prior work. Although space limitations prevent us from going into full detail, we provide some pointers here for the interested reader.

- *Measurements of availability*: The seminal work in availability data collection is Gray's study of Tandem computer system failures [Gray86]. More recent work include Murphy's availability studies of VAX systems and Windows 2000 [Murphy95] [Murphy00].
- *Recovery Experiments*: This topic is closely related to fault insertion, which to many means stuck-at faults in gates. It has a long history in the fault-tolerance community and covers a range of techniques from heavy-ion irradiation [Carreira99] to software simulation of hardware bugs [Segal88] [Arlat92] and programming errors [Chandra98] [Kao93]. Our FIG system inherits conceptually from work on system robustness testing using corrupt and/or exceptional inputs [Koopman00] [Siewiorek93], but differs in that it evaluates application, not OS, recovery. In this sense, the FIG technique follows the bottom-up approach advocated by Whittaker [Whittaker01]. Finally, most existing work uses fault insertion as an offline technique used during system development, although there are a few systems that have been built with the capability for online fault injection (notably the IBM 3090 and ES/9000 mainframes [Merenda92]).
- *Problem diagnosis*: There are several standard approaches to problem diagnosis. One is to use models and dependency graphs to perform diagnosis [Choi99] [Gruschke98] [Katker97] [Lee00] [Yemini96]. When models are not available, they can either be discovered [Kar00] [Brown01] [Miller95] [Zave98], or alternate techniques can be used, such as Banga's system-specific combination of monitoring, protocol augmentation, and cross-layer correlation [Banga00]. Our Pinpoint example demonstrates another approach by tracking requests through the system as is done in distributed resource utilization monitors [Reumann00].
- *Undo*: Our model of an "undo for operators," based on the three R's of rewind, repair, and replay, appears to be unique in its ability to support arbitrary changes during the repair stage. However, there are many similar systems that offer a subset of the three R's, including systems like the EMC TimeFinder [EMC02], a slew of checkpointing and snapshot systems [Elnozahy96] [Borg89] [Lowell00] [Zhou00], and traditional database log recovery [Mohan92]. Additionally, our undo implementation relies heavily on existing work in non-overwriting storage systems.

10. Discussion and Future Directions

"If it's important, how can you say it's impossible if you don't try?"
Jean Monnet, a founder of the European Union

We have presented new statistics on why services fail, finding that operator error is a major cause of outages and hence a major portion of cost of ownership. Following Peres' Law, ROC assumes that hardware, software, and operator faults are inevitable, and that fast recovery may improve both availability and cost of ownership. We argued that failure data collection, availability benchmarks involving people, and margin of safety will be necessary for success.

We hope that, by presenting a broad set of case studies, we have demonstrated the plausibility of ROC techniques, some of which were inspired by other fields. For pedagogical reasons, we presented five case studies as if each only exercised one technique. In reality, most case studies used several techniques, as Figure 9 shows. We think this is a good sign, as we hope the techniques are widely applicable.

We note the importance of recovery experiments in evaluating virtually all our proposals. Since they also enable the availability benchmarks, progress in ROC requires interfaces to processors, memories, I/O devices, and operating systems that support recovery experiments.

ROC looks hard now because as a field we are not positive we can do it. As the quote above suggests, however, if you agree that dependability and maintainability are important, how can we say it is impossible if we do not try? At this early stage, the ROC horizon is full of important challenges and opportunities:

- We need a theory of constructing dependable, maintainable sites for networked services. This theory will likely affect operating systems and the architecture of clustered computers.
- We need a theory of good design for operators as well as good design for end users. Using an airline analogy, its as though we have good guidelines for passengers but not for pilots. Operating systems designers have often ignored the impact of their decisions on operators, leading to high TCO. Another issue is users reporting errors in ways that will help operators repair systems.
- We need a more nuanced definition of failure than up or down, perhaps by looking at granularity of failure. Can we find the information technology equivalent of blocked calls (see Figure 2)?
- Performing recovery experiments in the field can train operators and remove latent errors, provided partitioning contains the scope of the experiment and undo erases mistakes in recovery. Do we need architectural/operating system innovation to enable live recovery experiments?
- We likely need performance monitors to show the first signs of misbehaving modules. Can these observations also improve performance in computers and operating systems?
- We need to economically quantify the cost of downtime and ownership. Without easy ways to measure them, who will buy new systems that claim to improve them?
- We need to continue the quest for real failure data and to develop useful availability and maintainability benchmarks. This quest helps measure progress, lower barriers to publication by researchers, and humiliates producers of undependable computer products (and research projects).
- The design of our initial email prototype is intentionally simplistic; it is primarily a testbed for examining policies governing externalized actions. In future versions, we intend to extend the prototype by providing undo on a per-user basis (to allow users to fix their own mistakes), by providing read-write access during the undo cycle by synthesizing consistent states from the information in the log, and by adding hooks to the mail server to reduce the proxy's complexity and improve the system's recovery time further. Our intent is that this prototype will eventually leverage all six ROC techniques.
- Given the difficulty of triggering faults in hardware, virtual machines may be worth investigating as a recovery experiment vehicle. Trusting a VM is more akin to trusting a processor than it is to trusting a full OS [CN01]. VMs may also help with partitioning and recovery time, as it may be plausible to have hot standby spares of virtual machines to take over upon a fault. Finally, a poorly behaving system can occupy so many resources that it can be hard for the operator to login and kill the offending processes, but VM provides a way out–an ideal topic for a future ASPLOS.

ROC technique	Mercury RR	FIG	Email with Undo	Pinpoint	ROC-1
Partitioning	√				√
Recovery experiments	√	√	√	√	√
Reversible systems			√		
Diagnosis aid				√	√
Independent mechanisms	√		√		√
Redundancy and fail fast					√

Figure 9. Six ROC techniques and their actual use in the five case studies of Sections 4 to 8.

Acknowledgements

We are supported at Berkeley on an NSF ITR and at Stanford on an NSF Career Award, plus industrial support from Allocity and Microsoft. For comments on early drafts of this paper, we wish to thank Howard Alt, Tome Anderson, Gaetano Borriello, Jim Gray, James Hamilton, Christoforos Kozirakis, Kai Li, Udi Manber, David Martin, Jeff Mogul, John Ousterhout, Scott Pyne, Yi-Min Wang, and John Wilkes.

References

[Arlat92] J. Arlat, A. Costes, et al. Fault Injection and Dependability Evaluation of Fault-Tolerant Systems. *LAAS-CNRS Research Report 91260*, January 1992.

[AUDITS83] 5ESS AUDITS Description and Procedures Manual, Issue 2, 1983.

[Baker92] M. Baker and M. Sullivan. The recovery box: Using fast recovery to provide high availability in the UNIX environment. *Proc. of the Summer USENIX Conf.*, June 1992.

[Banga00] G. Banga. Auto-diagnosis of Field Problems in an Appliance Operating System. *Proc. of the 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[Bobbio98] A. Bobbio and M. Sereno. Fine grained software rejuvenation models. *Proc. IEEE Int'l. Computer Performance and Dependability Symp.*, Sep. 1998, pp. 4–12.

[Borg89] A. Borg, W. Blau, W. Graetsch et al. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[Bres95] T.C. Bressoud and F.B. Schneider. Hypervisor-based fault tolerance. In Proc. Fifteenth ACM Symp. on Oper. Sys. Principles (SOSP-15), Copper Mtn., CO, Dec. 1995.

[Brewer 01] Brewer, E.A. Lessons from giant-scale services. *IEEE Internet Computing*, vol.5, (no.4), IEEE, July-Aug. 2001. p.46-55.

[BST02] Broadwell, P, Naveen Sastry Jonathan Traupman. Fault Injection in glibc (FIG), www.cs.berkeley.edu/~nks/fig/paper/fig.html.

[Brown00] A. Brown and D.A. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. *Proc 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[Brown01] A. Brown, G. Kar, and A. Keller. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. *Proc 7th IFIP/IEEE Int'l. Symp. on Integrated Network Management*, Seattle, WA, May 2001.

[Brown02] A. B. Brown and D.A. Patterson. Rewind, Repair, Replay: Three R's to Dependability. *Submission to 10th ACM SIGOPS European Workshop*, 2002.

[Brown02a] A. B. Brown, L. C. Chung, and D. A. Patterson. Capturing the Human Component of Dependability in a Dependability Benchmark. Submitted to the *2002 DSN Workshop on Dependability Benchmarking.*

[Candea01a] Candea, G.; Fox, A. Recursive restartability: turning the reboot sledgehammer into a scalpel. *Proc. 8th Workshop on Hot Topics in Operating Systems*, 2001. p.125-30.

[Candea01b] Candea, G.; Fox, A, Designing for high availability and measurability. *1st Workshop on Evaluating and Architecting System dependability.* Göteborg, Sweden, July 1, 2001.

[Candea02] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, R. Gowda. Minimizing Mean-Time-to-Recover in a Recursively Restartable Software System. To appear in *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002)*, Washington, DC, June 2002.

[Carreira99] J. Carreira, D. Costa, and J. Silva. Fault injection spot-checks computer system dependability. *IEEE Spectrum* 36(8):50-55, August 1999.

[Chandra98] S. Chandra and P. M. Chen. How Fail-Stop are Faulty Programs? *Proc. of the 1998 Symp .on Fault-Tolerant Computing (FTCS)*, June 1998.

[Chen96] P. M. Chen, W. T. Ng, S. Chandra, et al. The Rio File Cache: Surviving Operating System Crashes. In *Proc. of the 7th Int'l. Conf. on ASPLOS*, pages 74-83, 1996.

[Chen02] M. Chen, E. Kıcıman, et al. Pinpoint: Problem Determination in Large, Dynamic Internet Services. To appear in the 2002 DSN International Performance and Dependability Symposium (IPDS/DSN 2002), June 2002.

[CN01] Peter M. Chen and Brian Noble. When virtual is better than real. In Proc. Eighth Symp.on Hot Topics in Operating Systems (HotOS-VIII), Elmau, Germany, May 2001.

[Choi99] J. Choi, M. Choi, and S. Lee. An Alarm Correlation and Fault Identification Scheme Based on OSI Managed Object Classes. *1999 IEEE Int'l. Conf. on Communications*, Vancouver, BC, Canada, 1999, 1547–51.

[Elnozahy96] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *CMU Technical Report CMU TR 96-181*, Carnegie Mellon University, 1996.

[EMC02] EMC TimeFinder, http://www.emc.com/products/software/timefinder.jsp.

[Fox97] A. Fox, S. Gribble, Y. Chawathe et al. Cluster-based Scalable Network Services. *Proc. of the 16th Symp. on Operating System Principles (SOSP-16)*, St. Malo, France, October 1997.

[Fox99] Fox, A.; Brewer, E.A. Harvest, yield, and scalable tolerant systems. *Proc. of the 7th Workshop on*

Hot Topics in Operating Systems, Rio Rico, AZ, USA, 29-30 March 1999. p.174-8.

[Garg97] S. Garg, A. Puliafito, M. Telek, K.S. Trivedi. On the analysis of software rejuvenation policies. *Proc. of the 12th Annual Conf. on Computer Assurance*, June 1997, pp. 88–96.

[Gates02] Gates, W. Miscrosoft email, January 15, 2002. Reported in N.Y.times.

[Gillen02] Gillen, Al, Dan Kusnetzky, and Scott McLaron "The Role of Linux in Reducing the Cost of Enterprise Computing", IDC white paper, Jan. 2002, available at.

[Gray78] J. Gray,: *Notes on Data Base Operating Systems. Advanced Course: Operating* Systems 1978: 393-481.

[Gray86] J. Gray. Why Do Computers Stop and What Can Be Done About It? *Symp.on Reliability in Distributed Software and Database Systems*, 3–12, 1986.

[Gray96] J. Gray, P. Helland, P. O'Neil, and D. Shasa. The Dangers of Replication and a Solution. *Proc. of the 1996 ACM SIGMOD Int'l.Conf. on Data Management*, 1996, 173-182.

[Gray02] Gray, J. *What Next? A dozen remaining IT problems*," Turing Award Lecture, 1999

[Gribble00] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. *Proc. of the Fourth Symp.on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[Gruschke98] B. Gruschke. Integrated Event Management: Event Correlation Using Dependency Graphs. *Proc. of 9th IFIP/IEEE Int'l.Workshop on Distributed Systems Operation & Management (DSOM98)*, 1998.

[Hennessy99] J. Hennessy, The Future of Systems Research, *Computer*, August 1999 32:8, 27-33.

[HP02], Hennessy, J., D. Patterons, Computer Architecture: A Quatitative Approach, 3rd edition, Morgan Kauffman, San Francisco, 2002.

[Hitz95] D. Hitz, J. Lau, M. Malcolm. File System Design for an NFS Server Appliance. *Network Appliance Technical Report TR3002*, March 1995.

[Huang95] Y. Huang, C. Kintala, N. Kolettis, N.D. Fulton. Software rejuvenation: analysis, module and applications. *Proc. 25th Int'l.Symp.on Fault-Tolerant Computing*, June 1995, pp. 381–390.

[IBM 01]] IBM, Automonic Computing, http://www.research.ibm.com/autonomic/., 2001

[Jain88] Jain, A. and Dubes, R., () _*Algorithms for Clustering Data*. Prentice Hall, New Jersey. 1988.

[Kao93] W. Kao, R. Iyer, and D. Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, 1993.

[Kar00] G. Kar, A. Keller, and S. Calo. Managing Application Services over Service Provider Networks: Architecture and Dependency Analysis. *Proc. of the Seventh IEEE/IFIP Network Operations and Management Symp.(NOMS 2000)*, Honolulu, HI, 2000.

[Katker97] S. Kätker and M. Paterok. Fault Isolation and Event Correlation for Integrated Fault Management. *Fifth IFIP/IEEE Int'l.Symp.on Integrated Network Management (IM V)*, San Diego, CA, 1997, 583–596.

[Kembel2000] Kembel, R. *Fibre Channel: A Comprehensive Introduction*, p.8, 2000.

[Koopman00] P. Koopman and J. DeVale. The Exception Handling Effectiveness of POSIX Operating Systems. IEEE Transactions on Software Engineering 26(9), September 2000.

[Lahiri01] T. Lahiri, A. Ganesh, R. Weiss, and A. Joshi. Fast-Start: Quick Fault Recovery in Oracle. *Proc. of the 2001 ACM SIGMOD Conf.*, 2001.
Lampson, B. [1999] *Computer Systems Research-Past and Future*, Keynote address, 17th SOSP, Dec. 1999.

[Lee00] Lee, I.; Iyer, R.K. Diagnosing rediscovered software problems using symptoms. IEEE Transactions on Software Engineering, vol.26, (no.2), IEEE, Feb. 2000.

[LeFebvre01]. W. LeFebvre. CNN.Com: Facing a World Crisis. Keynote talk at the 15th Annual Systems Administration Conference (LISA 2001), San Diego, CA, December 2001.

[Leveson93] Leveson, N.G., Turner, C.: An investigation of the Therac-25 accidents. *IEEE Computer*, July 1993, page 18-41

[Lowell97] D. E. Lowell and P. M. Chen. Free Transactions with Rio Vista. In *Proc. of the 16th ACM Symp.on Operating Systems Principles*, October 1997, pp. 92-101.

[Lowell98] D. E. Lowell and P. M. Chen. Discount Checking: Transparent, Low-Overhead Recovery for General Applications. *University of Michigan CSE-TR-410-99*, November 1998.

[Lowell00] D. E. Lowell, S. Chandra, and P. Chen. Exploring Failure Transparency and the Limits of Generic Recovery. *Proc. of the 4th Symp.on Operating*

System Design and Implementation. San Diego, CA, October 2000.

[Merenda92] A. C. Merenda and E. Merenda. Recovery/Serviceability System Test Improvements for the IBM ES/9000 520 Based Models. *Proc. of the 1992 Int'l.Symp.on Fault-Tolerant Computing*, 463–467, 1992.

[Meyers77] M. N. Meyers, W. A. Routt, and K. W. Yoder. No. 4 ESS: Maintenance Software. Bell Systems Technical Journal, 56(7), September 1977.

[Miller95] B. Miller, M. Callaghan et al. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer* 28(11):37–46, November 1995.

[Mohan92] C. Mohan, D. Haderle, B. Lindsay et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1): 94–162, 1992.

[Murphy95] 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.

[Murphy00] B. Murphy and B. Levidow. Windows 2000 Dependability. *Microsoft Research Technical Report, MSR-TR-2000-56*, June 2000.

[Opp02] Oppenheimer, D. et al [2002]. ROC-1: Hardware Support for Recovery-Oriented Computing, *IEEE Trans. On Computers*, 51:2, February 2002.

[Oracle01] Oracle 9i Flashback Query. *Oracle Daily Feature*, August 13, 2001, available at technet.oracle.com/products/oracle9i/daily/Aug13.html.

[Perrow90] Perrow, Charles [1990], *Normal Accidents: Living with High Risk Technologies*, Perseus Books.

[Petroski92]. Petroski , H. *To engineer is human : the role of failure in successful design* /, Vintage Books,. NewYork, 1992

[Reason90] , Reason J. T.. *Human error*, New York : Cambridge University Press, 1990.

[Reumann00] J. Reumann, A. Mehra, K. Shin et al. Virtual Services: A New Abstraction for Server Consolidation. *Proc. of the 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[Rosenblum92] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *Proc. of the 13th Symp.on Operating System Principles*, pages 1-15, October 1991.

[Santry99] D. J. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch. Elephant: The File System that Never Forgets. In *Proc. of the IEEE Workshop on Hot Topics in Operating Systems (HOTOS)*, March 1999.

[Segal88] Z. Segall et al. FIAT—fault injection based automated testing environment. *Proc. of the Int'l.Symp.on Fault-Tolerant Computing*, 1988, pp. 102—107.

[Seltzer93] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. *Proc. of the Winter 1993 USENIX Conf.*, San Diego, CA, January 1993, 307-326.

[Siewiorek93] D. Siewiorek et al. Development of a Benchmark to Measure System Robustness. In Proceedings of the 1993 International Symposium on Fault-Tolerant Computing, 88-97, June 1993.

[Steininger99] A. Steininger and C. Scherrer. On the Necessity of On-line-BIST in Safety-Critical Applications—A Case-Study. *Proc. of the 1999 Int'l.Symp.on Fault-Tolerant Computing*, 208–215, 1999.

[Stonebraker87] M. Stonebraker. The Design of the POSTGRES Storage System. *Proc. of the 13th Int'l.Conf. on Very Large Data Bases (VLDB)*, September 1987.

[Whittaker01] J. A. Whittaker. Software's Invisible Users. IEEE Software, May/June 2001.

[Yemini96] S. Yemini, S. Kliger et al. High Speed and Robust Event Correlation. *IEEE Communications Magazine*, 34(5):82–90, May 1996.

[Zave98] Zave, P. Architectural solutions to feature-interaction problems in telecommunications. Proc. 5th Int'l Workshop on Feature Interactions in Telecom. Software Systems, Lund, Sweden,. 1998..

[Zhou00] Y. Zhou, P.M Chen, Kai Li Fast cluster failover using virtual memory-mapped communication. Conference Proc. 1999 Int'l Conf. on Supercomputing, Rhodes, Greece, 20-25 June 1999