Proceedings of 2000 USENIX Annual Technical Conference

San Diego, California, USA, June 18–23, 2000

TOWARDS AVAILABILITY BENCHMARKS: A CASE STUDY OF SOFTWARE RAID SYSTEMS

Aaron Brown and David A. Patterson

© 2000 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Towards Availability Benchmarks: A Case Study of Software RAID Systems

Aaron Brown and David A. Patterson
Computer Science Division, University of California at Berkeley
387 Soda Hall #1776, Berkeley, CA 94720-1776
{abrown,patterson}@cs.berkeley.edu

Abstract

Benchmarks have historically played a key role in guiding the progress of computer science systems research and development, but have traditionally neglected the areas of availability, maintainability, and evolutionary growth, areas that have recently become critically important in high-end system design. As a first step in addressing this deficiency, we introduce a general methodology for benchmarking the availability of computer systems. Our methodology uses fault injection to provoke situations where availability may be compromised, leverages existing performance benchmarks for workload generation and data collection, and can produce results in both detail-rich graphical presentations or in distilled numerical summaries. We apply the methodology to measure the availability of the software RAID systems shipped with Linux, Solaris 7 Server, and Windows 2000 Server, and find that the methodology is powerful enough not only to quantify the impact of various failure conditions on the availability of these systems, but also to unearth their design philosophies with respect to transient errors and recovery policy.

1 Introduction

There is a consensus emerging in parts of the systems community that the traditional focus on performance has become misdirected in today's world, a world in which the problems of availability, maintenance, and growth have become at least as important as peak performance, if not more so. One need only open up a recent issue of the *New York Times* or *Wall Street Journal* to see evidence of this fact—the number of stories focusing on recent outages of big e-commerce providers and the major business impact of those outages is staggering; furthermore, several of those outages have been reported as resulting from errors made by systems management staff [19]. Even from a financial standpoint, availability and manageability are important: the management costs for servers providing 24x7 service are typically reported as being several times that of the hardware itself [9][11][12].

The research community is beginning to recognize the importance of focusing on maintainability, availability, and growth as well. The attendees of the 7th HotOS workshop concluded that achieving "No-Futz Computing" (incorporating ideas of manageability, reliability, and availability, amongst others) is one of the most pressing challenges facing systems researchers today [20]. And, in his keynote at the 1999 FCRC conference, John Hennessy argued the same point, insisting that "performance should be less of an emphasis. Instead, other qualities will become crucial: *availability* [...], *maintainability*, [... and] *scalability*. [...] For servers—if access to services on servers is the killer app—availability is *the* key metric" [13]. Furthermore, the traditional "scalability problem," of creating and efficiently using large massively-parallel systems, is giving way to what we call the "evolutionary growth problem": constructing large-scale servers that can be incrementally expanded using newer, heterogeneous components.

Benchmarks have historically helped shape computer systems research and development, and we believe it is time for them to do so again. It is time for benchmarks to expand past the space of performance measurement and into the realm of quantifying availability, manageability, and growth; once such benchmarks exist, research into these areas will become significantly more tractable and research progress will naturally follow. Thus, as part of the Berkeley ISTORE project [4], we are investigating techniques for building reproducible, cross-platform "AME" benchmarks for Availability, Maintainability, and Evolutionary Growth, the three challenge areas laid out by Hennessy [13].

In this paper, we present our first steps toward that goal. We have chosen to focus initially on availability, and to begin by developing a general benchmarking methodology for measuring availability. As an initial proof of concept, we have applied this methodology to measure the availability of the software RAID-5 implementations that ship with three popular PC server operating systems: Linux, Solaris 7 Server, and Windows 2000 Server. RAID-5 is just the first example, and we hope our approach will inspire others to benchmark availability of other subsystems.

We chose software RAID as a case study for several reasons. First, software RAID implementations are included with many commercial OS releases (such as the server editions of Solaris 7 and Windows 2000) and with all of the major free UNIX-like operating systems, including Linux, which is being increasingly deployed for Internet service applications. More importantly, RAID has well-defined availability goals, making it an ideal candidate application for benchmarking availability. Also, it is not unusual to find software RAID underlying many Internet service applications that demand 24x7 availability, and thus the availability of the RAID implementation plays an important role in that of the service application itself. Finally, although there is agreement on general features of a RAID-5 system, availability benchmarking can highlight RAID implementation decisions that are important to applications but that are not measured or even mentioned today, for example how a RAID system distinguishes between a disk failure and a temporary glitch.

In studying the availability of the software RAID systems, we found significant differences in implementation philosophy between the various OS implementations. The major differences in philosophy between the systems can be classified along two axes: the first measures the system's paranoia with respect to transient errors, while the second measures the relative priorities placed on preserving application performance versus quickly rebuilding redundancy after a failure. On these axes, the Linux software RAID implementation is paranoid about transients but values application I/O performance more than fast post-failure reconstruction. Solaris falls at the opposite end of both spectrums, demonstrating a near-complete tolerance for transient errors and emphasizing fast reconstruction despite its potential impact on application performance. Windows 2000 falls between Linux and Solaris, although it lies closest to the Solaris end of the spectrum: it tolerates a set of transient errors that is only slightly less robust than Solaris's, and demonstrates a reconstruction philosophy that is similarly aggressive but more workload-aware than Solaris's. The fact that our benchmarks could reveal these philosophies despite treating the implementations as black boxes highlights the power of the methodology.

The remainder of this paper is organized as follows. First, we describe our generic methodology for availability benchmarking in Section 2. In Section 3, we show how that methodology was specialized for the case of measuring software RAID availability, and describe our experimental approach. We present our availability results and describe our experience with the benchmarks and RAID systems in Section 4. Section 5 discusses related work, Section 6 presents our future plans for this work, and we conclude in Section 7.

2 A General Methodology for Availability Benchmarking

In this section, we describe a general methodology that can be used to measure and study the availability of arbitrary computer systems. We begin by establishing a standard definition of availability and the metrics that can be used to report it, then consider how to construct benchmarks that produce those metrics, and finally describe how the results of those benchmarks can be reported and analyzed.

2.1 Availability: definitions and metrics

The term "availability" carries with it many possible connotations. Traditionally, availability has been defined as a binary metric that describes whether a system is "up" or "down" at a single point of time. A traditional extension of this definition is to compute the percentage of time, on average, that a system is available ("up") or not ("down")—this is how availability is defined when a system is described as having 99.999% availability, for example.

We take a different perspective on availability. First, we see availability as a spectrum, and not a binary metric. Systems can exist in a large number of degraded, but operational, states between "down" and "up." In fact, systems running in degraded states are probably more common than "perfect" systems [2], especially in the fast-growing world of online service provision where economic pressures encourage deployment of less-well-tested commodity SMP- and cluster-based servers rather than expensive fault-tolerant machines. An availability metric must therefore capture these degraded states, measuring not only whether a system is up or down, but also its efficacy, or the quality of service that it is providing.

Second, availability must not be defined at a single point in time or as a simple average over all time. It must instead be examined as a function of the system's quality of service over time. To motivate this, consider that from a user's perspective, there is a big difference between a system that refuses requests for two seconds out of every minute and one that is down for one whole day every month, even though the two systems have approximately the same average uptime. Any benchmark of availability must be able to capture the difference between those two systems.

Combining these two requirements, we propose that availability be measured by examining the variations in system quality of service metrics over time. The particular choice of quality of service metrics depends on the type of system being studied. Two obvious metrics that apply to most server systems are performance and degree of fault-tolerance. For a web server, these metrics would map to requests satisfied per second (or per-

haps latency of request service) and the number of failures that can be tolerated by the storage subsystem, network connection topology, and so forth. Other possible metrics might include:

- *completeness*: consider a system like the Inktomi search engine that tolerates failures by returning search results that cover only the remaining available parts of its database [10];
- *accuracy*: a system that must perform a large computation in a fixed amount of time (e.g., decoding real-time media) might sacrifice accuracy in the computation when running in degraded mode; and
- *capacity*: to maintain other metrics while in a degraded state, a system might limit the number of clients or jobs it will accept, or might discontinue less-essential services.

We discuss how these time-dependent availability measurements might be concretely represented as graphs and numerical summary statistics in Section 2.3, below.

2.2 Towards an availability benchmarking methodology

Having selected the availability and quality-of-service metrics for a given type of system, our next challenge is to accurately and reproducibly measure them in a controlled benchmarking environment. Doing so is complicated by the fact that typical benchmark environments are explicitly designed to prevent the kinds of exceptional behavior that would cause availability to be affected in real-world systems.

Thus, in order to perform availability benchmarks, it is necessary to have a benchmark environment that provides a means of generating fault-provoking stimuli and "*maintenance events*" and applying them to the system under test. (A maintenance event is any action taken by a human administrator to maintain, repair, or upgrade the system.) The primary technique that enables such an environment to be constructed is direct *fault injection* into the system under test [1][5]. For example, disk failures in a storage array can be simulated, memory can be artificially corrupted, processes can be killed, power glitches can be simulated, network links can be broken, and so forth. Fault injection need not be limited to hardware faults, however: stimuli such as load spikes, invalid client/user requests, and other workload-driven ways of triggering boundary conditions are also reasonable events to simulate.

To build an availability benchmark, we also need a way to generate a realistic workload and to measure the appropriate quality of service metrics. Our task is simplified by leveraging the extensive efforts at fair workloads from the performance benchmarking community. We simply use existing performance benchmarks to generate a representative workload for the type of system under test, and to measure the desired metrics at a single point in time. These workload-generating performance benchmarks should be adapted to run continuously, repeatedly measuring the desired metric. The system under test may also need to be modified to measure certain metrics (such as accuracy or completeness).

Given a benchmark environment supporting fault injection and a performance benchmark configured as both a continuous workload generator and a quality of service data collector, running an availability benchmark consists of two steps. First, the workload generator is run without injecting faults and several traces of the values of the desired metrics are recorded. This step establishes a baseline measurement for a non-faulty system. Second, the workload generator is run while simultaneously injecting a *fault workload*, and again a trace of the values of the desired metrics is recorded. This second step is key, since it produces a trace of the behavior of the system's quality of service over time in response to various faults, which is exactly the time-dependent availability metric that is desired.

The only part of the methodology we have not yet discussed is the content of a "fault workload". As its name suggests, a fault workload is a collection of faults and maintenance events designed to mimic a real-world failure situation.

We see the need for two different kinds of fault workloads, described in the following two sections, roughly corresponding to traditional micro- and macro-benchmarks:

Single-fault workloads. The first kind of fault workload is the availability analogue of a performance microbenchmark. A single-fault workload, as its name implies, consists of just a single fault: once the system under test has reached steady-state, a single fault is injected—such as a disk sector write error—and the system's behavior (as reflected in the quality of service metrics) is recorded. Intervention of a human administrator in response to the fault is not allowed. Like performance microbenchmarks, single-fault availability benchmarks are most useful for studying isolated pieces of a system and for uncovering design decisions, design flaws, and bugs. Their scope is broader than performance microbenchmarks, however, since a single fault can often have a ripple effect and affect a system as much as a multi-fault workload.

Multi-fault workloads. The second kind of fault workload is the availability equivalent of a performance macrobenchmark. Multi-fault workloads consist of a series of faults and maintenance events designed to mimic real-world fault scenarios, for example, a disk failure in a RAID system followed by replacement of the failed

disk followed by a write failure while reconstructing the array. Like traditional application performance macrobenchmarks, multi-fault workloads are useful for building availability benchmarks designed to help select or evaluate new systems, and to identify potential weaknesses in existing systems that need to be addressed. They are also very useful for studying the behavior of the system under pathological failure conditions (as in the RAID example above).

A challenging problem in developing benchmarks based on multi-fault workloads lies in how to realistically and reproducibly simulate the behavior of a human administrator in maintaining the system and in responding to failures originating from fault injection. Such "maintenance events" cannot be ignored, as very few modern systems are truly self-maintaining and most will require human intervention to complete the scenarios. We believe that the solution is to use logs of administrator activity to develop a stochastic model of the system maintenance process and of how administrators typically behave in response to various system failures and stimuli. For example, one might characterize the distribution of response time between a reported disk failure and the replacement of that disk. Such a model can then be used to direct the human intervention in the system during the benchmark run. There is a parallel here to performance benchmarks designed for systems that require human interaction; often in these benchmarks, a script plays back what a person would type in response to prompts. We are currently pursuing this approach.

Note that disk improvements over the years mean that disks no longer fail fast: the classic head crash of operating systems lore almost never happens today, as disks have become physically smaller and their mean time between failures has increased from 50,000 to 1,000,000 hours. Observations of the Tertiary Disk (TD) system at UC Berkeley, a large disk and web server farm, suggest that modern components start acting erratically rather than failing fast, and so a system administrator is much more likely to "fire" and replace an erratic component than to wait for it to fail completely [23]. We feel it is important to capture this type of activity in any model of administrator behavior.

2.3 Analyzing and reporting availability benchmark results

The raw data produced from either a single-fault- or multi-fault-workload availability benchmark is rather unwieldy, and therefore some standard techniques for analyzing and reporting it are required.

The simplest way to handle the data from the runs with fault injection is to plot it graphically, with the quality of service metrics on the vertical axis and time on the horizontal axis. The graph is then overlaid with confidence intervals calculated from the runs in which no faults were injected; these intervals indicate the range of quality of service values that are statistically "normal." Finally, the times at which faults were injected are marked on the graph. An example of this type of graph is shown in Figure 1.

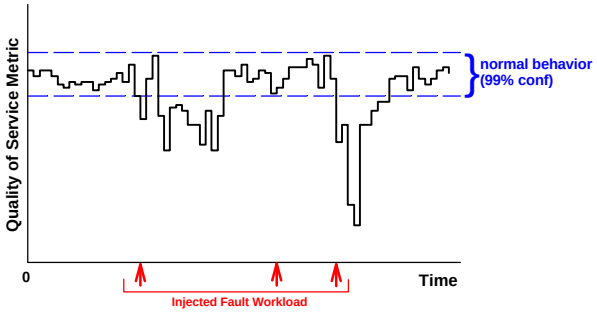

Figure 1: Example availability graph, showing the variation in an application quality of service metric (on the vertical axis) over time (on the horizontal axis), as faults are injected into the system (the faults are represented by heavy arrows). The dashed lines define a 99% confidence interval around the system's normal (non-faulty) behavior.

These graphs provide a good means by which the experimenter or system designer can study and understand the availability behavior of the system, and they are what we will use later in this paper to report our results for software RAID. In particular, the experimenter can use these graphs to focus on the points at which the measured values of the quality of service metrics fall outside the statistically normal range; these are the points where the system's availability has been compromised.

However, the graphs remain somewhat difficult to quantify and compare, especially if the benchmarks are to be used by end-users or customers. Several SPEC benchmarks do report graphs, and some customers do compare the graphs side-by-side. But the salient features of the graphs can also be distilled numerically. The most direct approach here is to identify all deviations from the statistically normal range, and to characterize—via mean, standard deviation, and possibly a distribution function—the distributions of the frequency of those deviations, the length of those deviations, and the severity (height) of those deviations. By characterizing the distribution rather than just averaging, this approach may preserve, for example, the distinction between the system that is down 2 seconds every hour and the one that is down one day every month. Of course, these characterizations can be distilled further, for example by simply reporting the product of the average length and average severity of the deviations, although at this point the benchmark result begins to lose much of its descriptive power.

3 Implementing the Methodology for Software RAID

In the previous section, we presented a general methodology for benchmarking system availability. In this section, we describe how we implemented that methodology for measuring the availability of the software RAID implementations provided by Linux, Solaris 7 Server, and Windows 2000 Server.

The availability guarantees of RAID-5 are straightforward [7]. A RAID-5 volume can tolerate a single disk failure without loss of data. After that first failure, the volume can continue to service requests in "degraded" mode, although I/Os tend to be more expensive due to the need to reconstruct data on-the-fly. A second disk failure renders the data on the volume inaccessible. Some RAID-5 implementations support spare disks, and can restore redundancy by rebuilding onto the spare after the first failure; during this reconstruction period, the volume will still be destroyed if a non-spare disk fails, although failure of the spare disk can be tolerated.

3.1 Fault injection environment

For the experiments in this paper, we chose to limit the fault injection to faults affecting the disks comprising the software RAID volume, as those are the primary hardware failure points in a software RAID system. Since we wanted to generate a range of different disk faults in a controlled manner, we rejected the simplistic fault-injection technique of pulling disks out of a live system. Instead, we replaced one of the SCSI disks in the software RAID volume with an *emulated disk*, a PC running special software with a special SCSI controller that makes the combination of PC+controller+software appear to other devices on the SCSI bus as a disk drive (*i.e.*, a SCSI target rather than a SCSI controller). Thus our systems under test saw the PC emulating the disk as a real disk drive.

Our emulated disk consisted of an AMD-K6-2-350 PC with an ASC ASC-U2W SCSI adaptor, running Windows NT with the ASC VirtualSCSI Target Mode Emulation library installed [3]. We adapted the library to emulate one or two SCSI disk drives by converting I/O requests to the emulated disk into reads and writes to two large backing files on a dedicated local disk on the emulation machine. The files holding the contents of the emulated disks were the only files on the local disk, only one file/emulated disk was active at once in any given experiment, and all accesses to the backing files passed through the NTFS file system layer but bypassed the buffer cache. The emulation layer added a constant overhead of approximately 510 microseconds to each disk I/O. Compared to a Linux file system on one of the real disks used in our RAIDs, this emulation overhead translates to 10% fewer seeks per second, 41% less write bandwidth, and 16% less read bandwidth, as measured by the 100MB Bonnie benchmark.

More importantly, we modified the disk emulator to allow the injection of faults into the emulated disk. To make our benchmarks as realistic as possible, it was essential that our set of injected disk faults closely match the types of disk faults seen in practice. To that end, we turned to a study performed as part of the aforementioned Tertiary Disk project at UC Berkeley. Using the 368 disks in the TD array, Talagala recorded the types of faults that occurred over an 18-month period [23]. She found that the most common errors and failures affecting disks included recovered (media) errors, write failures, hardware errors (such as device diagnostic failures), SCSI timeouts, and SCSI bus-level parity errors.

Using this set of errors as a guide, we selected several categories of faults to include in our emulator:

- *correctable media errors* on reads and writes, to simulate disk sectors starting to go bad;
- *uncorrectable media errors* on reads and writes, to simulate unrecoverably-damaged disk sectors;
- *hardware errors* on any SCSI command, to simulate firmware or mechanical errors;
- *parity errors* at the SCSI command level, to simulate SCSI bus problems;
- *power failures* that simulated a disk being disconnected, both during and between SCSI commands;
- *disk hangs* that simulated disk firmware bugs/failures both during and between SCSI commands (these appear as SCSI timeouts to the controller).

All of the faults (except for the fatal ones, like simulating disk power down or infinite timeout) could be inserted either in transient mode, in which case they appeared once then disappeared, or in sticky mode, in which case they continued to manifest themselves once injected. We were particularly interested in the behavior of the software RAID systems in response to the transient faults, as results from Talagala's TD study indicate that disks rarely fail fast, but rather tend to die slowly with an ever-increasing number of transient and correctable faults [23]. Most availability guarantees made by RAID systems speak only of discrete failures, not of such "fail-slow" failures.

As desired, our set of injectable faults closely matches the set of error conditions seen in the TD array. Note that we were unable to inject one of these types of error condition with our fault-injection harness: the SCSI parity errors at the level of the SCSI electrical protocol. Simulating this type of fault requires either direct access to the wires of the SCSI bus or to low-level registers within the controller, neither of which were available to us.

3.2 Configuration of systems under test

We examined three software RAID implementations in our experiments, those shipped with Linux, Solaris 7 Server with Solstice DiskSuite, and Windows 2000 Server. In all cases, the OS and RAID system were installed on a PC with an AMD-K6-2-333 CPU, 64MB of 66MHz ECC DRAM, and a Seagate 5400RPM IDE system disk. Three physical SCSI disks were attached to the machine. Each disk was an IBM DMVS18D 18GB 10000RPM Ultra2-LVD SCSI low-profile drive. Each drive was connected to its own dedicated Fast/Wide (20 MB/s) SCSI bus. Each of the three busses was terminated by either an Adaptec 2940UW controller (set to Fast mode) or one channel of an Adaptec 3940W controller. Note that each drive had a private SCSI bus that was not shared with any other device. A 1GB partition was created at the beginning of each drive for use in the experiments; the remainder of the space on each drive was unused.

The emulated disk (*i.e.*, the PC running the emulation software) was connected to a fourth dedicated SCSI bus on the machine under test using an Adaptec 2940UW controller. Two 1GB emulated disks were created; one was used in the RAID and the other was left as a spare (thus the two were never simultaneously part of the active RAID volume). The backing files for the emulated disks were placed on a dedicated NTFS file system at the beginning of a dedicated IBM DMVS18D 18GB 10000RPM Ultra2-LVD SCSI low-profile drive.

In all cases, unless otherwise noted, the software RAID volume was a 3GB-capacity RAID-5 volume encompassing the three physical disks and one emulated disk. When supported by the implementation, the second emulated disk was used as a spare.

For Linux, we used the Redhat 6.0 distribution with version 0.90-3 of the RAID tools. The RAID volume was configured as 4 active disks plus one spare, left-symmetric parity, and a chunk size of 32. An ext2 file system was used with a 4KB block size and a stripe width of 8.

The Solaris system ran the 3/99 release of Solaris 7 for Intel architectures. We installed version 4.2 of Sun's Solstice DiskSuite and used it to create a RAID-5 "metadevice" with 4 active disks and one spare. The RAID volume was formatted with a Solaris UFS file system with default parameters.

The Windows 2000 Server system was running release candidate build 2128 of the operating system. We used the supplied volume manager to create the RAID-5 logical volume out of 4 active disks. Windows 2000 does not support automatic spares, as far as we could determine, so the spare disks were left as separate dynamic disks in the volume manager. An NTFS file system was used on the array with default parameters.

The three systems were configured as web servers with the documents served from the RAID volume and the logs written to the RAID volume. The Linux and Solaris systems used Apache 1.3.9 as the server, while Windows 2000 Server used the included version of Microsoft's IIS as the server.[1] Other than relocating the logs and document directories to the RAID volume, the servers were left in their default configurations. IIS's performance configuration knob was set to "more than 100,000" hits per day.

3.3 Workload generator and data collector

In order to complete our experimental testbed, we needed a source of workload for the web servers running on each OS, and a means of continuously measuring the quality of service delivered by the web servers over time. We chose to use SPECWeb99 [22], a standard web performance benchmark, for both of these tasks. SPECWeb99 uses one or more clients to generate a realistic, statistically reproducible web workload; its workload models what might be seen on a busy major server, and includes static and dynamic content, form submissions, and server-side banner-ad rotation. In each iteration, the benchmark applies a load designed to elicit a certain aggregate bandwidth from the server, then measures the percentage of that bandwidth that was actually achieved. It also measures the number of hits per second delivered by the server and the average response time; we chose to use the number of hits per second (a throughput-oriented performance metric) as the quality of service metric as it was the most tractable and because the other metrics tracked it relatively closely.

We modified the workload generator slightly so that it would fit our model of continuous performance measurement over time: we removed all warm-up and cool-down periods other than the initial warm-up period, adjusted the per-iteration time to 2 minutes, and set the number of iterations to a very large number (manually stopping the generator when the benchmark was complete). This allowed us to obtain performance measurements every two minutes, with each number reflecting the average performance over the previous two-minute period.

We also adjusted the workload generator to reduce the amount of dynamic content from 30% to 1% to keep the disks busy and to avoid saturating the CPU. This restriction was necessary because we used the default high-overhead perl-cgi implementation for dynamic

1. We chose to use IIS for Windows 2000 rather than Apache as we wanted to select the web server that would be typically used with each OS. Since IIS ships with Windows 2000 Server, we believed that it would be the appropriate choice.

content and the CPU on our server testbed was not able to keep up with the higher level of dynamic content.

We configured the applied workload to be just short of the saturation point on each of the three systems by increasing the number of active connections per second (the SPECWeb99 load unit) until a knee was observed in the performance curve, then backing off the load by 5 connections per second. The three systems each saturated at different points, and thus we applied a different level of load to each system in our tests; this accounts for the differences in absolute performance that show up in Figures 2 and 3, below. We chose this load profile instead of applying a consistent load to all three machines in order to isolate the worst-case availability impact on each system. This profile also ensures that we were making fair comparisons between the systems, as some availability behavior (such as RAID reconstruction speed) can be affected by the amount of free system resources. Where pertinent, we also discuss results from experiments in which the applied load was reduced to below the saturation point on each system.

Finally, note that we observed heavy disk activity during the benchmark runs on all three systems, indicating that server-side caching effects were not significant.

4 Results

In this section, we present the results of applying our availability benchmarking methodology to the software RAID implementations provided by Linux, Solaris, and Windows 2000. We first look at the single-fault availability microbenchmarks, then move on to study more complex multi-fault availability macrobenchmarks.

4.1 Single-fault microbenchmarks

Recall that single-fault microbenchmarks involve injecting a single fault into a running system and observing the resulting behavior of that system without any human intervention. To perform these microbenchmarks for the software RAID systems, we first configured the RAID volume to its nominal state: all disks working, and all spares available. We then started the SPECWeb99 workload generator and allowed it to reach steady state. We next injected a single fault, and allowed the system to continue running (collecting performance data) until the system recovered (performance returned to its steady-state level), stabilized at a different performance level than its steady-state level, or failed. We define failure as not providing service for at least 10 iterations (20 minutes) with no apparent signs of ever returning to service.

In all cases, the faults that we injected were chosen to affect active disk blocks, guaranteeing that the system would be aware of them. By doing so, we avoid injecting so-called *latent* faults, faults that cannot cause failures since they affect only unused data or control paths. We feel this is a reasonable policy for an availability microbenchmark, as the goal of such benchmarks is to characterize the system's response to specific faults, and not to measure susceptibility to randomly-placed faults.

We injected a total of 15 types of faults, listed in Table 1. Each fault-injection experiment was repeated at least twice, and in all cases, similar behavior was observed in each iteration. In our experiments, we found no evidence of silent corruption from any injected fault. All faults that could potentially result in corrupted data were either detected by the OS's disk driver or RAID layer. What differentiates the systems is not their detection abilities, but their behavior in response to the detected faults.

Surprisingly, these response behaviors across the three systems and the 15 types of injected faults can be classified into only five distinct categories, also listed in Table 1. Representative availability graphs for each of these categories are plotted in Figure 2. We classify two of the behaviors (**C-1** and **C-2**) as subcategories of the same major behavior category, as they represent the same response behavior (automatic reconstruction) but differ in their performance characteristics. Note that each graph in Figure 2 plots the change in two metrics with respect to time. The first metric, represented by a solid line, is the same performance metric discussed above: the number of hits per second delivered by the web server running on the system under test, averaged over two-minute intervals. The second metric, represented by a broken line, represents the minimum number of disk failures the system is theoretically able to tolerate; it is effectively a measure of the system's data redundancy. Note that the graphs also show 99% confidence intervals that were computed from the traces of the systems' normal no-fault performance.[2]

Of the four major categories of observed behavior, the first, **A** in Figure 2, represents the behavior pattern that occurs when an injected fault has no effect on the RAID system. This graph plots the behavior of the Solaris system in response to a transient, correctable read fault. Notice that the performance curve remains within the confidence intervals despite the injection of the fault; the redundancy measure remains unchanged as well. Effectively, the Solaris system ignores this fault, as it is essentially benign; the disk correctly satisfied the read request, but needed to use ECC bits or multiple reads to obtain the data. Both the Solaris and Windows 2000 systems displayed behavior of this type. Solaris responded this way to all non-fatal faults that we injected, including transient uncorrectable faults (such

2. Analysis showed that the no-fault performance data was normally distributed; thus, the 99% confidence intervals were computed as 2.576 sample standard deviations on either side of the sample mean.

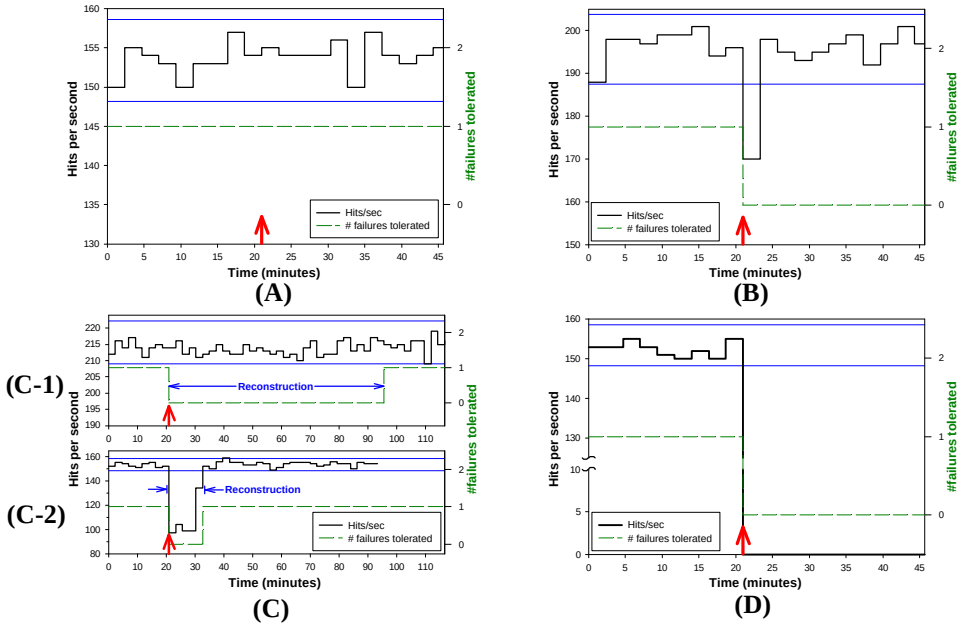

Figure 2: Representative availability graphs displaying the five different patterns of behavior observed after injecting faults into the three software RAID systems. Each graph plots two metrics: on the left vertical axis, and represented by a solid line, is the number of hits per second sustained by the web server on the system under test, reported as a single average value over each two-minute interval. On the right vertical axis, and represented by a broken line, is the theoretical minimum number of disk failures the system should be able to tolerate without losing data. Fault injection points are represented by heavy arrows, and 99% confidence intervals for the normal (non-faulty) behavior of the systems are defined by the thin horizontal lines. Table 1 maps each type of injected fault into one of these five behaviors (**A**, **B**, **C-1**, **C-2**, **D**) for Linux, Solaris, and Windows 2000.

as a transient, non-repeatable write failure). Windows 2000 behaved similarly, although it was slightly less tolerant of write errors (it did not exhibit this behavior pattern for transient uncorrectable write faults). In no cases did Linux exhibit pattern **A**—it never transparently tolerated a non-fatal fault.

The second category, **B** in Figure 2, is more complicated. In this case, the fault is severe enough that the RAID system stops using the affected disk, but is not so severe that the RAID system cannot tolerate it. The performance is slightly affected only during the interval in which the fault was injected, as the system detects and recovers from the fault. The redundancy curve indicates that the faulty disk is no longer used: in this case, the system does not automatically rebuild onto a spare disk, and thus the system cannot tolerate any more disk failures. The particular data plotted in Figure 2(B) is the behavior of Windows 2000 in response to a simulated power failure on one disk of the array (equivalent to physically pulling an active drive from a hot-swap array). This pattern also characterizes Windows's response to other severe faults, including sticky uncorrectable read faults and all uncorrectable write faults.

The magnitude of the performance drop during the fault-injection iteration depended on the type of fault; for uncorrectable writes, it was about 4% of the mean performance, and for power failures, it was about 13% of the mean. Note that the performance drop during the

Type of Fault	Behavior		
	Linux	Solaris	Win2k
Correctable read, transient	C-1	A	A
Correctable read, sticky	C-1	A	A
Uncorrectable read, transient	C-1	A	A
Uncorrectable read, sticky	C-1	C-2	B
Correctable write, transient	C-1	A	A
Correctable write, sticky	C-1	A	A
Uncorrectable write, transient	C-1	A	B
Uncorrectable write, sticky	C-1	C-2	B
Hardware error, transient	C-1	A	A
Illegal command, transient	C-1	C-2	A
Disk hang on read	D	D	D
Disk hang on write	D	D	D
Disk hang, not on a command	D	D	D
Power failure during command	C-1	C-2	B
Physical removal of active disk	C-1	C-2	B

Table 1: Classification of system behavior for each of the injected faults. The letters in the rightmost three columns correspond to the pattern of behavior observed after the specified fault is injected, as shown in Figure 2.

fault-injection iteration occurs because the server is near saturation. If we reduce the applied load by just over 20%, the observed performance drops become statistically insignificant. This indicates that Windows is able to trade spare resources for reduced availability impact in certain failure scenarios.

Neither Solaris nor Linux exhibited pattern **B**, as they both support automatic recovery onto a spare disk: when the Solaris or Linux software RAID driver detects a fault severe enough to stop using a disk, it immediately begins reconstructing the data from the failed disk onto the available hot spare. This pattern is illustrated in the graphs labeled **C-1** and **C-2** in Figure 2. **C-1** plots Linux's response to a transient correctable read fault, and **C-2** plots Solaris's response to a sticky uncorrectable write error.

In the Solaris case, we see that the performance curve drops significantly below the lower bound of the confidence interval during the reconstruction period. In contrast, Linux's performance during its entire reconstruction period is statistically indistinguishable from its unperturbed performance. However, Solaris completes reconstruction significantly faster than Linux. The significance of these behavioral differences will be discussed further when we compare the reconstruction behavior of Solaris and Linux with Windows's non-automatic reconstruction in Section 4.2.

Note that during reconstruction, the redundancy curve is not well-defined; the system cannot tolerate a fault to any of the data disks, but it can tolerate a fault to the spare (the destination of the reconstruction).

While Solaris exhibited its version of pattern **C** only for three of the 15 faults (two of which were unquestionably fatal faults), Linux exhibited pattern **C-1** for every injected fault but those falling into pattern **D** even if the fault was transient and non-fatal (like a correctable read).

Finally, the last category, **D**, represents what happens when the RAID system is unable to tolerate the injected fault. As can be seen, the performance drops to zero when the fault is injected; this is usually a result of the RAID driver or operating system hanging. The redundancy curve is not well-defined in this case, since the system is not operational. We observed this type of fault in Solaris, Linux, and Windows when we injected particularly pathological disk hangs in the middle of SCSI command execution.

Table 1 summarizes how the 15 types of injected faults map to the five categories of behavior for each of the operating systems.

Analysis. Although limited to a single fault each, these microbenchmark results reveal very interesting facts about the availability guarantees of Linux, Solaris, and Windows 2000; none of these facts were stated in the documentation supplied with the three systems. Most illuminating are the conclusions that can be drawn about how the three systems treat transient faults. If we exclude the pathological disk hangs and power-failure faults, 8 of the remaining 10 injected fault types simulate transient or recoverable errors that in isolation do not indicate immediate disk failure. Four of these 8 do not even require that the corresponding I/O's be retried. The remaining two faults (sticky, uncorrectable reads and writes) are the only faults in the set of 10 that indicate that the disk is in an unrecoverable state.

Yet for every fault in this set of 10 non-pathological faults, the Linux system exhibited behavior of type **C**, in which the faulty disk is immediately removed from service. In contrast, both Solaris and Windows kept the faulty disk in service on 7 of the 10 non-pathological faults (*i.e.*, 7 of the 8 recoverable errors). Solaris disabled the faulty disk (pattern **C-2**) upon the two unrecoverable faults (sticky uncorrectable reads/writes) as well as on a transient illegal command fault. This behavior is arguably slightly more robust than that of Windows, which disabled the faulty disk (pattern **B**) upon the two unrecoverable errors and a transient uncorrectable write, since an illegal command error typically implies a coding error in the driver or a serious disk firmware error, rather than a potentially transient magnetics glitch.

From these observations, we can conclude that Linux's software RAID implementation takes a totally opposite approach to the management of transient faults than do the RAID implementations in Solaris and Windows. The Linux implementation is paranoid—it would rather shut down a disk in a controlled manner at the first error, rather than wait to see if the error is transient. In contrast, Solaris and Windows are more forgiving—they ignore most transient faults with the expectation that they will not recur. Thus these systems are substantially more robust to transients than the Linux system. Note that both Windows and Solaris do log the transient errors to varying extents, ensuring that the errors are reported even if not acted upon. Windows is more explicit with its reporting, for example visually flagging a disk as "at risk" in the RAID management GUI upon a correctable write error, whereas Solaris relies on the system log for its error recording.

We cannot draw conclusions about a RAID system's overall robustness based solely on its transient-error-handling policy, however. There is another factor that interacts with a system's error handling, and that is its policy for reconstruction. The microbenchmarks demonstrate that both Linux and Solaris initiate automatic reconstruction of the RAID volume onto a hot spare when an active disk is taken out of service due to a fail-

ure. Although Windows supports RAID reconstruction, the reconstruction must be initiated manually, as discussed further in Section 4.2, below. Thus without human intervention, a Windows system will not rebuild redundancy after a first failure, and will remain susceptible to a second failure indefinitely.

The policy choice of automatically or manually-initiated reconstruction interacts strongly with the transient error-handling policy in affecting system robustness. A paranoid RAID implementation without hot spares is very fragile, as it takes only two transient errors to corrupt the RAID volume; likewise, an indifferent RAID implementation has less of a need for hot spares as it will only stop using a disk upon a serious fault. Thus in our case, the non-robustness of the Linux implementation's paranoid approach to transients is mitigated somewhat by its automatic reconstruction, and similarly Windows's lack of automatic reconstruction is partially mitigated by its robustness to transients. Solaris seems to combine the best of both: robustness to transients plus automatic reconstruction upon a fatal error.

Returning to the three systems' transient error policies, if we consider these policies in the context of real failure data, such as that gathered by the Tertiary Disk project, it is clear that none of the observed policies is particularly good, regardless of reconstruction behavior. Talagala reports that transient SCSI errors are frequent in a large system such as the 368-disk Tertiary Disk farm, yet rarely do they indicate that a disk has truly failed [23]. Tertiary Disk logs covering 368 disks for 11 months indicate that 13 disks reported transient hardware errors, yet only two actually required replacement. Those two did not "fail-fast" with head crashes, either: both were replaced due to an excessively large number of transient errors. Additionally, due to the effect of shared SCSI busses and at-times flaky SCSI cabling, at some point over that period every disk in the system was involved in some sort of SCSI error (such as a parity error or timeout) [24]. Even if we ignore these SCSI errors and focus only on the transient hardware errors, Linux's policy would have incorrectly wasted 11 real disks (3% of the array) and potentially 11 spares (another 3% of the array) due to its over-zealous reaction to transient errors. Even worse, if the array did not have enough spares to keep up with the disk turnover, data could have been lost despite the fact that no disk truly failed. Equally poor would have been the response of Solaris or Windows 2000, as these systems most likely would have ignored the stream of intermittent transient errors from the two truly defective disks, requiring administrator intervention to take them offline.

A better RAID implementation would have a more balanced policy for dealing with transient errors. For example, it might be less paranoid initially, tolerating transient faults until they reached a certain frequency or absolute count, at which point the system would declare a disk dead and stop using it (note that our macrobenchmark experiments showed that neither Windows nor Solaris did this). This kind of policy balances the need for long-term availability (which favors a more relaxed policy) with the fact that disks tend to fail with a stream of transient errors rather than failing fast.

Although none of the RAID implementations we examined is ideal, we can conclude from the microbenchmarks that either Solaris's or Windows 2000's RAID is more suitable for applications requiring high long-term data availability, as both are less likely to fall prey to multiple transient errors (especially in systems that are not closely monitored or conscientiously administered). However, for applications where spare disks are plentiful and short-term availability is most important (*i.e.*, when the performance impact of many transient errors cannot be tolerated, when the system is closely monitored, and when repairs are made quickly), the Linux implementation may be a better choice.

Our results and analysis also argue strongly for the importance of exposing the policy decisions that affect availability in systems like these software RAID implementations. Ideally, the policies would be made configurable, for example by allowing the administrator to select a point on the spectrum between Linux's paranoid response to transients and Solaris's tolerance of them. Doing so would make the policies explicit, and may even simplify maintenance of the system by increasing its predictability, thereby eliminating the need for the administrator to guess at how the system will behave under various conditions.

At the very least, availability policies such as those governing the system's response to transient errors should be documented so that administrators and buyers can evaluate the potential robustness of their systems in their particular environment. Until such documentation is commonplace, availability benchmarks such as those described here may well remain the only way to identify and evaluate these important but well-concealed policies.

4.2 Multiple-fault macrobenchmarks

After measuring the effects of single failures on the availability of the Linux, Solaris, and Windows software RAID implementations, we next constructed two "fault workloads" designed to mimic real-world scenarios and applied them to the three systems.

Scenario 1: Reconstruction. The first scenario includes five events, and models a situation in which a nominally-configured RAID-5 volume with one spare (1) experiences a failure on one of its active disks, (2) is

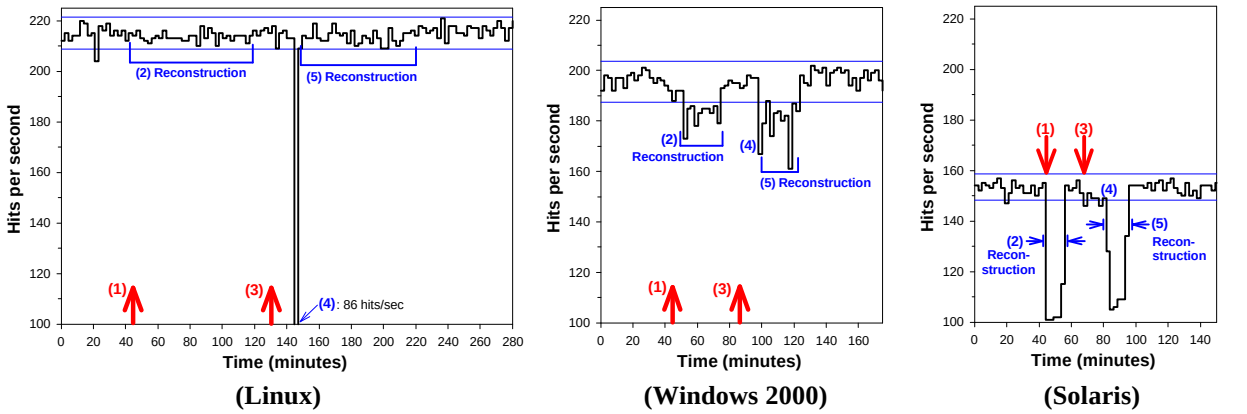

Figure 3: Availability graphs for an availability macrobenchmark with a multiple-fault workload. On the vertical axis, and represented by a solid line, is the number of hits per second sustained by the web server on the system under test. The change in this metric is plotted versus time on the horizontal axis. The thin horizontal lines represent the 99% confidence interval defining the system's normal (no-fault) behavior. The two injected faults are indicated by heavy arrows. The numbers in parentheses on each graph indicate the corresponding part of the fault scenario, as described in Section 4.2. The absolute performance differences between the three systems are due to different applied loads, as described in Section 3.3.

reconstructed (automatically or manually) using the spare, and (3) later experiences a failure on the then-active spare. The scenario is finished by (4) the administrator replacing the two failed disks and (5) reconstructing the volume's redundant data onto one of the new disks. The behaviors of Linux, Windows 2000, and Solaris on this macrobenchmark are plotted in Figure 3. Note that for Windows, we inserted a 6-minute delay to simulate sysadmin response time between detecting the first failure and manually starting the reconstruction. The process of "replacing" the broken (simulated) disks was performed manually, and took approximately 90 seconds in each case.

One obvious difference between the behaviors of the three systems on this benchmark is that Linux and Solaris automatically reconstruct whereas Windows requires human intervention. Most interesting is the difference in reconstruction time between the three systems, and in the performance impact of reconstruction in each case. Linux is the slowest to reconstruct, taking well over an hour each time. However, there is no significant effect on application performance during reconstruction; other than during the time that the disks were being replaced, the performance curve does not fall outside of the confidence interval for normal behavior while reconstruction is taking place.

Solaris defines the opposite extreme. Its reconstruction is over 7 times faster than Linux's, lasting just over 10 minutes. But this speedy reconstruction comes at a performance cost: the web server performance on Solaris is below the lower bound of its normal behavior for the entire reconstruction interval, with a maximum deviation of 34% from its mean no-fault performance.

Windows's behavior is similar to Solaris although not as extreme. Its reconstruction lasts approximately 23 minutes, over twice as slow as Solaris but still more than three times faster than Linux. Windows too shows a performance drop during reconstruction, but it is less significant than Solaris's: the worst-case performance observed was only about 18% below the no-fault mean.

From these observations we can conclude that Solaris and Windows are dedicating more disk bandwidth to reconstruction than is Linux. This again reveals a design tradeoff in the three systems that would be difficult to detect without benchmarks such as these: Linux chooses to emphasize preserving application performance over speedy reconstruction, even though it sacrifices short-term availability. In contrast, Solaris puts a high priority on restoring redundancy despite the performance impact. Windows makes the same tradeoff toward prioritizing reconstruction, but does so less aggressively than Solaris.

Another interesting characteristic of reconstruction is how this behavior changes as the load on the system is reduced. While space constraints prevent us from providing a full analysis in this paper, we did find that at lower loads (such that the systems were unsaturated), Linux and Solaris each exhibited the same reconstruction behavior as in the saturated case in terms of reconstruction time and performance impact. In contrast, Windows was able to decrease both its reconstruction time and the impact of reconstruction on application performance. Our hypothesis is that these behaviors are a function of the scheduling discipline in each of the OSs as well as the priority each system assigns to the reconstruction task. The implication of these behaviors is again significant for availability: Windows seems to be the only system of the three that is able to use the excess resources resulting from lower imposed load to mitigate the availability impact of reconstruction. In

practice, this means that Windows is the only system of the three that can take advantage of hardware with a higher saturation point to improve its availability characteristics as well as its performance potential.

The differences in reconstruction philosophy revealed by this benchmark once again argue for the importance of exposing policies that affect availability. The three RAID systems examined here offer very different robustness guarantees because of their undocumented reconstruction policies. We saw this above in how Windows compared to the other systems under reduced load. Another example is that Linux, with its slow, low-priority reconstruction, has a much larger window of vulnerability to double failures, a weakness exacerbated by its susceptibility to transient errors. This policy is unsuitable if data integrity is most important, and in that case a policy like Solaris's is a better choice. On the other hand, if delivering consistent application performance is more important than preserving the data at all costs, then Linux's policy is reasonable and Solaris's unacceptable. An ideal system would offer the administrator a spectrum of choices between these two extreme policies, but we feel that every system should at least document its chosen policy. Benchmarks such as these offer a convenient tool for doing so.

Scenario 2: Double failure. The second scenario mimics a catastrophic failure in RAID systems reported anecdotally by multiple sources. The scenario begins when a nominally-configured RAID volume (1) experiences a disk failure that causes the faulty disk to be removed from service and (2) begins reconstruction (automatically or manually). At that point, (3) the well-meaning system administrator attempts to replace the failed disk, but accidentally pulls out the wrong disk—one of the remaining live disks rather than the dead one; (4) he or she then tries to restore the system to a working state. Removing the live disk should result in a catastrophic failure of the RAID volume, although it did not do so in all cases, as we discuss below. The graphs up to this point are relatively uninteresting, confirming the expected behavior, and are not reproduced here.

What is interesting is the behavior of the systems after the catastrophic failure, and the difficulty of restoring service on the system. We describe this behavior only qualitatively, since in order to quantify it, we would have to find some way of measuring the maintainability of the system, perhaps by modeling the length and complexity of the repair task, which is beyond the scope of this paper.

In this scenario, the last, "catastrophic" failure is actually reversible. According to the RAID availability semantics, the RAID volume should stop serving requests upon a double failure. If the RAID implementation queues writes to the removed disk while it is unavailable, the administrator could put the disk back in, and theoretically, the system should be able to recover. We tested this hypothesis on the three systems in order to see how close each of them came to this theoretical possibility.

Windows 2000 actually came remarkably close, although it does not queue writes to disconnected disks. After reactivating the accidentally-removed disk (which required a few GUI operations), Windows allowed the RAID volume to be accessed despite its possibly corrupt state, and the web server resumed serving requests to the SPECWeb99 clients. Running CHKDSK as recommended revealed no file system corruption (probably due to the journaling nature of NTFS). Since the web workload was essentially read-only except for the log writes, the only data lost was logging information.

In contrast, we found it impossible to resurrect the Linux RAID volume. The tool used to reintegrate a disk into the volume seemed to only be capable of adding new disks to the volume as spares, which are then automatically used as the target of a reconstruction. There was no obvious way to use the existing tools to convince Linux that the replaced disk contained real data. Therefore, the only way to resurrect the volume was to recreate and reformat it, then restore data from backup.

Solaris demonstrated radically different behavior than the other two systems. Unlike the other systems, it did not disable the RAID volume after the double failure: it kept the array active with the two still-functioning disks and the partially-reconstructed spare. This behavior violates the availability semantics of RAID-5, since at this point a large portion of the data is missing (any data that had not yet been reconstructed on the spare is permanently lost). By keeping the RAID array active and using the nonsensical data on the partially-reconstructed spare, Solaris allows applications to read garbage data. In our case, this was manifested by the web server returning garbage to the SPECWeb client and via numerous UFS file system corruptions as reported by fsck. Furthermore, when we plugged the accidentally-removed disk back in, Solaris was happy to automatically switch back to using it to service I/Os, deactivating the partially-reconstructed spare. However, because Solaris had continued to use the array while the second disk was removed, the data on that disk was significantly out-of-date and the file system was corrupted as a result of reinserting it.

We believe that Solaris's behavior is absolutely incorrect for a RAID system. A RAID system should not fabricate data to maintain availability unless explicitly requested to do so, *i.e.*, by manually forcing the reactivation of a reinserted disk, as with Windows. Furthermore, we were not able to find any mention of this

behavior in any of the Solaris documentation, which again argues for the importance of benchmarks like these to expose the undocumented availability policies in systems like these software RAIDs.

Thus in this scenario, Solaris clearly loses due to its willingness to transparently serve up garbage data. But Windows 2000 wins on maintainability, as its robust file system and flexible RAID implementation allows the opportunity for at least some use of the RAID volume to continue servicing user requests while the system is being restored from backup (but only at the explicit request of the administrator, unlike Solaris). Although this may not always be the best thing to do, Windows provides the ability should it be desired.

In this second macrobenchmark, we have the beginnings of a framework for a combined availability and *maintainability* benchmark—the fault injection workload for this scenario brings the system to a state in which maintenance is required; to complete the benchmark, we would use a quantitative maintenance model to simulate repair of the system, then use that data to complete the availability graph for this scenario. We are currently pursuing this as future work.

5 Related Work

The notion of benchmarks to measure system availability or "robustness," although perhaps not familiar to the systems community, has not been neglected by the fault-tolerance community. Siewiorek describes "robustness benchmarks" based on fault injection performed primarily by using an application to feed corrupt input to the system [21]. Tsai, working on Tandem machines, proposes another set of reliability benchmarks based on software-implemented fault injection and a synthetic workload generator. His metrics include an average measure of performance degradation due to faults, a primitive version of our time-dependent quality of service metrics [25]. Koopman describes benchmarks to test OS robustness by feeding corrupt data to system calls [17]. The major difference between these benchmarks and the ones we propose is in their goals and the knowledge they assume. Tsai's and Siewiorek's benchmarks are primarily designed to test particular known fault-tolerance mechanisms deployed in fault-tolerant hardware and software systems; to this end, their benchmarks target and evaluate specific components, layers, or mechanisms in the system under test, and thus assume knowledge about the error-detection mechanisms and general structure of that system. In contrast, our benchmarks take a more black-box approach, assuming little about the system under test (not even that it is fault tolerant), and applying faults designed to match real-world failure patterns. Koopman's benchmarks do this as well, but are limited to faults generated by passing corrupt data to system calls; we try to mimic more general faults, including hardware failures.

An additional key difference is that our benchmarks measure the system's availability behavior in terms of application-specific metrics that reflect quality of service visible from the client's point of view. Finally, our multi-fault workloads go beyond the isolated faults examined by Siewiorek, Tsai, and Koopman by relating the behavior of a system to realistic scenarios that affect large-scale server systems and by providing a foundation for the expansion of the benchmarks to incorporate the measurement of maintainability.

The techniques of fault injection that we use are also not uncommon in the fault-tolerance community, where fault injection is commonly used in a case-specific manner to verify fault tolerant systems, to generate models of fault tolerance behavior, and to study fault propagation [1][5][6][8][16][18]. However, most of this work uses either very low-level hardware fault injection that requires expensive and dangerous equipment (such as heavy-ion bombarders) [5], or software-implemented fault injection. The former is not tractable for general use because of the cost and complexity, and the latter is not particularly portable, as it generally requires modifications to the OS or driver layer. In contrast, our approach of hardware fault injection at standard interfaces (such as the SCSI-level fault injection used for the RAID study) is both portable and relatively simple; for example, we could have easily used the same fault-injection setup (consisting of off-the-shelf PC hardware and software) to measure the availability of software RAID on a SPARC/Solaris machine.

Finally, there have been several studies of RAID reliability and availability [14][15], but these have focused on simulation studies of hardware RAID, and none have examined RAID in the context of a general availability benchmark.

6 Future Directions

We are currently pursuing several extensions of the work in this paper. First, we are planning to expand our experience with the availability benchmarks by applying them to more complex systems, such as database management systems. We are also working on a general framework for maintainability benchmarks, and in particular are looking into ways to model the behavior of a human administrator. We are also expanding the fault-injection capabilities of our testbed to include the capability of inserting memory faults and OS driver faults. Finally, under the umbrella of the ISTORE project, we are building the ISTORE-1 prototype, an 80-node cluster system that incorporates custom fault-injection and diagnostic hardware that should enable the extension of this work to distributed systems and applications.

7 Conclusion

In this paper we have laid out the framework for new kinds of benchmarks in an area left relatively unexplored by computer science researchers: availability. We demonstrated the efficacy of our general availability benchmarking methodology by specializing it to the study of software RAID systems, and by then using it to unearth insights into the behavior of the Linux, Solaris, and Windows 2000 software RAID implementations. In particular, we were able to uncover each system's (undocumented) policy for mapping transient faults into failure conditions, and to quantify the impact of these policies and of the systems's failure recovery policies on the quality of service and availability delivered by I/O-intensive applications running on those systems.

While we believe that the power of our approach is clearly illustrated in these insights, this paper is only a first tentative step down what surely must be a long road to the important goal of comprehensive, portable, and meaningful benchmarks for availability, maintainability, and evolutionary growth. We feel that reaching that goal is crucially important for the field, and we look forward to companionship on this journey.

Acknowledgments

This work was supported in part by ARPA grant DABT63-96-C-0056. The first author was supported by a Department of Defense National Defense Science and Engineering Graduate Fellowship. The germ for many of the ideas in this paper came out of discussions with members of the ISTORE group at UC Berkeley, and in particular with David Oppenheimer. We also wish to thank IBM for donating the disks used in these experiments, Andataco (and particularly Darryl Keiser) for providing extra drive enclosures on very short notice, Bill Casey of ASC for fixing the last bugs in the disk emulation library, and both the anonymous reviewers and members of the ISTORE group for their feedback. Finally, the first author thanks Randi Thomas for the encouragement to make this paper a reality.

References

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

[2] R. Arpaci-Dusseau. Performance Availability for Networks of Workstations. *Ph.D. Dissertation*, U. C. Berkeley. December, 1999.

[3] ASC, Inc. Advanced Storage Concepts VirtualSCSI library. http://www.advstor.com/vscsi.html.

[4] A. Brown, D. Oppenheimer, *et al.* "ISTORE: Introspective Storage for Data-Intensive Network Services." *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, Arizona, March 1999.

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

[6] S. Chandra and P. Chen. How Fail-Stop are Faulty Programs? In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, June 1998.

[7] P. Chen, E. Lee, et al. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys* 26(2):145–185, June 1994.

[8] R. Chillarege and N. Bowen. Understanding Large System Failure—A Fault Injection Experiment. In *Proceedings of the 1989 Fault-Tolerant Computing Symposium (FTCS)*, 356–363, 1989.

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

[10] A. Fox, S. Gribble, et al. Cluster-Based Scalable Network Services. In *Proceedings of SOSP '97*. October, 1997, St. Malo, France.

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

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

[13] J. Hennessy. The Future of Systems Research. *IEEE Computer* 32(8):27–33, August 1999.

[14] Y. Huang, Z. Kalbarczyk, and R. Iyer. Dependability analysis of a cache-based RAID system via fast distributed simulation. In *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.

[15] M. Kaâniche, L. Romano, et al. A Hierarchichal Approach for Dependability Analysis of a Commercial Cache-Based RAID Storage Architecture. In *Proceedings of 28th International Symposium on Fault Tolerant Computing*, June 1998.

[16] W. Kao, R. Iyer, and D. Tang. FINE: A Fault-Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults. *IEEE Trans. Software Eng.*, 19(11):1105–1118, November 1993.

[17] P. Koopman, J. Sung, et al. Comparing Operating Systems Using Robustness Benchmarks. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, 72–79, October 1997.

[18] W. Ng and P. Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proceedings of the 1999 Symposium on Fault-Tolerant Computing*.

[19] M. Richtel. Keeping E-Commerce On Line; As Internet Traffic Surges, So Do Technical Problems. *The New York Times*, 21 June 1999.

[20] M. Satyanarayanan. *Digest of Proceedings, Seventh IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*. March 29-30, 1999, Rio Rico, AZ.

[21] D. Siewiorek, J. Hudak, 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.

[22] SPEC, Inc. *SPECweb99 Benchmark.* http://www.spec.org/osg/web99.

[23] N. Talagala. Characterizing Large Storage Systems: Error Behavior and Performance Benchmarks. *Ph.D. Dissertation*, U. C. Berkeley. September, 1999.

[24] R. Thomas, N. Talagala. What Happens Before a Disk Fails. Talk at the Winter 1999 IRAM Semi-annual Research Retreat. January, 1999.

[25] T. Tsai, R. Iyer, and D. Jewett. An Approach towards Benchmarking of Fault-Tolerant Commercial Systems. In *Proceedings of the 1996 Symposium on Fault-Tolerant Computing (FTCS)*, June 1996.