Experiences in Measuring the Reliability of a Cache-Based Storage System

Dan Lambright
EMC
Dlambrig@emc.com

Abstract

We present our experiences in benchmarking the reliability of the cache component of a storage system in a development environment. The reliability metrics we measured are availability from the standpoint of the host and maintainability from the standpoint of the system operator. We created errors using software fault injection, and measured their impact using a combination of performance measurement techniques and the rehearsal of maintenance procedures. This paper gives three case studies. The first two describe experiments that recreate very specific breakdowns in the software logic, and the third describes an experiment simulating a memory hardware failure that creates unpredictable effects. We found that, taken together, these various techniques gave us a useful picture of how well our cache management software tolerated faults.

1. Introduction

Measuring the reliability of software, a relatively unexplored topic within the systems community, has recently been dubbed one of the major challenges facing computer scientists who come from that background [1].

From the standpoint of industry, the positive feeedback that reliability measurements bring is clear. Doing so helps determine whether one version of software is more reliable then a different version on the same system. It can alert developers to where attention should be focused, in particular whether valuable time should be spent writing a complex or simple solution. Creating errors also acts as a trial run for the recovery process employed by the system operator or customer service engineer, testing the robustness and effectiveness of diagnosis utilities for locating and fixing the problem.

In this paper, we discuss our experiences in measuring the reliability of the cache component of a modern disk array. We did not attempt to benchmark the entire system, choosing instead to focus on the cache because it lies at the heart of the architecture of the product. Errors frequently appear in cache before they appear on the disk [6]. The techniques we describe are by no means revolutionary. However, our goal is to demonstrate the usefulness of the techniques, and to give some insights into where future research could be directed.

In the product we examined, the error detection and recovery software of the cache subsystem is quite sophisticated. The design has no single point of failure and a great deal of resiliency in its structure. This work is intended to work torwards developing techniques to check if the hardware and software satisfies that design goal. Additionally, we desire a more explicit mechanism for determining how the software's effectiveness improves as it evolves. Presently, deriving that notion can be done by studying a dispersed set of statistics ranging from the binary results generated from regression testing, to reports summarizing the errors generated at customer sites. The desire for reliability benchmarks stems from a need to have a more immediate, reproducible, source of information.

The product we examined had software and hardware specifications readily available and modifiable for the purpose of our tests. This allowed us to employ software fault injection, a widely used technique, to generate faults [2][9][12].

The fault's impact on availability was in part measured with tools and techniques used by our performance measurement group. The fault's impact on maintainability was assessed by rehearsing the formal procedures a system operator would have taken in case of the fault, and noting what happened. We created faults to test both narrow, targeted points in the software logic ("targeted faults") and at broader problems ("untargeted faults"). The former technique was useful for stressing important algorithms in a very specific way. Realistically, such tests could not be custom designed for all the algorithms on the machine, and only provided reliability information on a small portion of code. For the system in its entirety, we turned to the later technique. Untargeted faults provide a reliability metric at the granularity of the system rather then the algorithm. By using both techniques selectively, we were able to get a more complete picture of reliability.

This paper is organized as follows: Section 2 characterizes the systems we benchmark. Section 3 discusses our methodology for generating faults and measurements. Section 4 contains studies showing how “targeted” faults test a narrow (but important) aspect of the system’s reliability. We found faults that acted randomly in a scattershot means provide better statistics on overall system robustness. Section 5 discusses those experiences. Section 6 explores some ideas relating to the integration of reliability benchmarking into software labs and future directions in research. Section 7 reviews related research, and section 8 concludes.

2. The Storage Device and Cache Subsystem

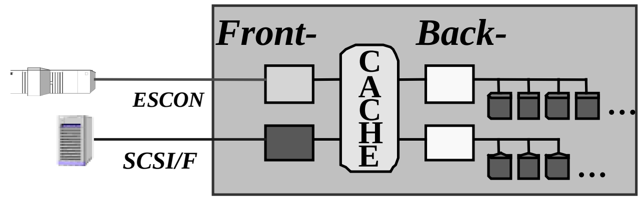
In this section, we give a very high level description of the architecture of the system we test and the role the cache plays in the system. We then describe in more detail how faults which appear in the cache are detected and categorized.

The product is a large disk array. The storage may be accessed over multiple I/O channels, which may be connected to varying types of hosts (e.g. MVS, UNIX), that consequently use varying protocols (e.g. ESCON, SCSI). A set of processors that we collectively call the “front-end” are devoted to interpreting the protocols and communications from the host systems. Another set of processors (the “back-end”) operates the disk drives. It consists of a set of CPUs that are divided between the disks according to configuration. The CPUs may communicate with each other via one or more networks, or a centralized memory, or both.

A large cache in the product serves as a holding place for data between the front and back-end. The cache is accessed from the CPUs over redundant, high-speed backplanes. On write operations, the front end CPUs transfer data from the channel to the cache, and the back end CPUs that control the target disks transfer data onto storage. On read operations, the path flows in reverse. Data in cache that has not yet been written to disk is “dirty”. Data is removed from cache using a variant of the LRU algorithm.

The system cache size may change dynamically. For example, if a given threshold of errors is detected, the cache is dynamically fenced off so that it no longer can be used. Additionally, the user may add, remove, or replace units of memory with minimal impact on availability to the user.

High Level Block Diagram of System



Errors in the cache are grouped into two categories: hardware and software. A software fault is a “bug” that resides in the code running on the front or back-end CPUs. A hardware fault consists of a number of bits which no longer function properly during read or write accesses within a given cache line. To be categorized as a hardware error, the number of faulty bits must exceed that which is correctable by the hardware’s EDAC (error detection and correction) logic. Hardware faults in the cache may also be manifested in faulty components (e.g. the front end or backplane malfunctions). There is redundancy at each level of the hardware to mitigate the impact of such problems.

Cache errors are further categorized by the impact they have on the user. For example, availability faults hinder performance but do not corrupt data. A hardware error that corrupts meta-data related to the LRU algorithm might affect optimal replacement strategy, but the user would still be able to load data at some reduced rate or response time.

Other metrics used to describe cache errors include latency and burstiness. Latency quantifies how long ago the fault occurred. Clearly, the shorter the latency, the easier it is to detect root cause, especially if the amount of space devoted to logs is limited. Burstiness describes what errors cluster together. A cache error may propagate to other errors on the disk or channel controller, or may be a symptom of a problem originating in those subsystems.

The system tracks errors using logs that preserve the context of the fault (e.g. stack trace, counters) for debugging. Errors are detected in the cache by background scrubbing tasks and during the I/O operations. The frequency of running the scrubbing software (which impacts error detection latency) must be balanced with providing enough time for processing host requests.

The cache has a set of diagnostic and development tools used to monitor aspects of the subsystem, including the state of the LRU algorithm, host utilization, meta-data associated with the cache (e.g. software locks), and error detection and recovery. The proper functioning of these diagnostic capabilities, even in the presense of severe faults, is important.

In summary, we are verifying the reliability of one of the components of a highly available system. We note that the system is made up of many different redundant subsystems, each of which could be analyzed separately. It would be useful to analyze the entire system as a whole, but the variety of hardware would make this a more difficult project.

3. Methodology

In this section, we describe the test configuration for our reliability measurements. Because of limited resources we had to trade off accuracy in our measurements for expediency. Nevertheless, the configuration was successful in helping us reach conclusions. We then discuss the metrics we use to measure availability and maintainability. For maintainability, our measurements were based in part on the subjective analysis of human beings. This is because of the complexity involved in developing automated measurement techniques.

To quantify availability, we adapted the general methodology for availability benchmarking to our environment [1]. Essentially, this procedure works by injecting one or more faults into the system while measuring a Quality of Service (QOS) metric. In our case, we wished to know the fault's impact on overall response time and throughput. These metrics appealed to us because they were already well-established in our vocabulary (for describing performance), and we had mature techniques and instruments, as well as seasoned in-house specialists, to do the measurements. Additionally, we knew of several applications (routinely used for our functionality tests), that were sensitive to unexpected deviations in those metrics.

Our workload generator consisted of a dedicated MVS mainframe running scripts to generate I/O. The storage system was connected to the host over 12 ESCON channels. The storage system had 8 GB of cache. There were 96 physical hard drives on the machine, each with a capacity of 18GB. The drives were partitioned into 288 "logical volumes," which were the "disks" visible to the host. We performed no other I/O or special applications on the storage subsystem.

For each test we ran a mix of 25% writes and 75% reads on randomly chosen blocks on the disks. This was a crude approximation of customer behavior, and ideally, we would prefer an I/O stream that predicted the impact on customers by mirroring actual user behavior. However, the behavior of users varies greatly from application to application. Creating a single profile representing the "average user" is beyond the scope of this experiment. In performance testing, different tests are run to test different classes of common I/O profiles (e.g. online transactions, sequential write). This will most likely be our course of action for future work.

We generated I/O requests from the host at a single rate, which was at a relatively low level compared to the maximum the system could handle. We took this approach in order to approximate the level of I/O of a typical system, rather then the level of an "envelope test", which might never be seen outside of performance benchmarking labs. Our measurement software, originally designed for performance testing, recorded the response time and I/O throughput once every minute. Our availability measuring tools were not accurate, but this was acceptable, as we were more interested in understanding the fault's impact then getting a high degree of precision.

The effect on maintainability was evaluated by manually simulating the corrective actions that the customer service engineer would take in the case of a fault. Those actions were known by following the instructions corresponding to the error in a knowledge database used by customer engineers, as well as interviewing them in person. When no solution in the knowledge database matched a fault, we determined what a customer engineer would do by conducting interviews and following our own judgement.

We considered writing an automated script to perform maintenance functions (derived by viewing logs of error recovery situations that had occurred in the past), but concluded this method was not helpful. A typical "solution" to a problem, as carried out by the system operator, involves a sequence of decisions, which are manifested by entering different diagnostic commands depending on the state of the machine at the time. A log only contains a sequence of commands for one unique situation. Simply "playing back" the same sequence of commands during fault injection could be inappropriate depending on the state of the machine at the time of the fault.

For example, suppose a fault in the cache was the type that propagated to another subsystem, such as the back-end CPU. The technician's first job would be to work on those problems, which would include checking the integrity of the disk. If the disk was a member of a RAID group, the data may be in several intermediate states. A script that accounted for the myriad possible states the machine could be in would be extremely complex and potentially error prone.

Maintainability was quantified using three parameters. Each parameter was rated on a scale of increasing quality: low, medium, and high. We call the first parameter effectiveness of diagnosis tools. To obtain it we noted the correct existence and operation of diagnosis utilities and error information. For example, in some cases we found that the faults we created did not have utilities to diagnose the problem. To obtain information the technician would be forced to have a relatively deep understanding of the code and dump raw memory and interpret it. In such cases the effectiveness of the tools would be judged to be of lower quality.

The second parameter was simplicity of solution. For example, a simple recovery would be to invoke a software correction utility, and a more difficult recovery would be to physically replace a component that failed, or upgrade the code. It is almost always preferable to employ the former solution.

Our third parameter, robustness of diagnosis tools, quantifies the correct functionality of management interface software in response to severe problems in the storage system; severe problems in the latter should not cascade into the former. Systems under intense stress should still be capable of interpreting such utilities and maintaining logs, otherwise the problem would persist, and there would be no corrective option other than shutting down the system.

Lastly, the degree of severity of the generated fault was tunable. This allows the tester to gradually increase the severity until the effect becomes noticeable, or the system completely shuts down. This “point of no return” is a useful data-point, even if it is completely unrealistic and would never happen under real circumstances.

In addition to availability and maintainability metrics, we recorded: (1) how long it took for the problem to be detected by the system, (2) whether it was self correcting, (3) whether it was streaming (i.e. recurring repeatedly).

To sumarize, we had little trouble finding existing techniques to measure availability, but found that obtaining measurements to measure maintainability had to be invented. A criticism of our maintenance tests may be that because our opinions on the “quality” of maintainability are subjective, they are subject to controversy. But there are inherent difficulties in automating maintenance tests. Finding practical solutions to those problems may be an interesting area of research.

4. Targeted Fault Injection

In this section, we describe the “targeted” tests we performed to measure the resilience of particular code modules and specific recovery paths. Our first case study examines a situation where an important data structure is not synchronized between the different CPU’s, and our second describes the effect of a rogue CPU that holds a shared software lock for unacceptably long periods.

4.1 Case Study 1: Unsynchronized Memory Maps

The first fault was a synchronization error. We wanted to test the behavior of the system when the different CPUs in the front and back end did not see the identical map representing which portions of the cache were available or fenced. In effect, some number of CPUs would believe that more memory existed than others. We believed such a problem would manifest itself in transient cache errors, but depending on the severity of the problem the number of errors may impact the overall availability of the system.

We wrote fault injection software to purposely break the synchronization of the memory maps. We adjusted the severity by changing the number of out-of-sync CPUs, and the size of cache (expressed in 32 MB chunks) disagreed upon. We attempted to break synchronization between both the front and back end CPUs to learn what difference that made in availability. We hypothesized that if the front-end CPU had more memory visible it would fill it with data on write operations, and the backend CPUs would then post an error when the dirty track was detected. Conversely, if the backend CPU had more memory, it would fill it on read operations, and the error would be posted immediately as the front-end CPU responded to the host.

For faults of low severity, we found no measurable effect on performance no mater which CPUs were out of sync. However, when we increased the number of out-of-sync CPUs to half those in the system, performance was noticeably impacted. Figures 1 and 2 illustrate the difference in impact between low and high serverity tests when backend CPUs were modified to have more memory visable. The respecting figures show the system’s response

Figure 1: I / O Rate for 1 Gig Out of Sync on 1 backend CPU

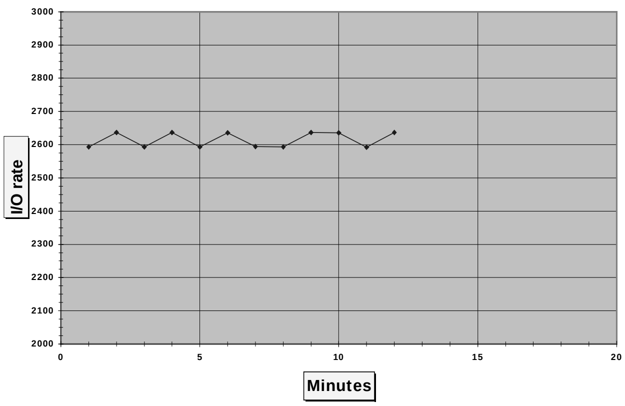
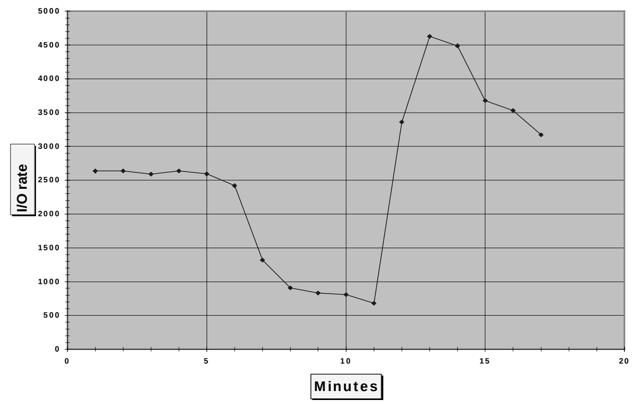


Figure 2: I / o Rate for 4 Gig Out of Sync; all backend CPUs



time over a 20 minute interval (displayed on the x and y axi, respectively), in which a synchronization fault was injected and repaired. In the low severity test, no effect on response time was noticed. In the high severity test, the fault was injected at minute 5 and corrected at minute 10. At minute 12, the response time jumped as the system apeared to catch up with requests that had been delayed.

The degredation in response time occurred because one part of the system would attempt to access disabled memory, generating an error. Each time this happened there was a small delay to report the error. The greater the severity the more the delays aggregated, hence this recovery period grew longer as the severity increased. We did not see cascading errors at low severity, but we did at high severity.

We found that at high severity, after we fixed the problem (by re-synchronizing the memory maps using diagnostic utilities), there was a brief period of continued performance degradation, the cause of which we are investigating. We also found that when we repeated the same high severity test on front-end CPUs, the impact on throughput was somewhat greater, which was in accord with our hypothposis that front-end CPUs are more vulnerable to this type of problem.

Maintainability in the synchronization case was attainable. The diagnostic procedure and problem discovery process was to manually check the memory map on each CPU, compare that with others, then disable the memory banks until all CPUs saw the same memory map. The transient errors did not affect management software functionality even at the greatest severity. We therefore rated robustness to be of high quality. However, the diagnostic utilities only showed the memory maps for individual CPUs, rather then all of them at a time, and they did detect the problem (i.e. they did not verify identical memory maps on each CPU). We therefore rated the diagnostic effectiveness to be of medium quality.

One concern was that to resynchronize the memory maps the CPU had to undergo a subset of the initialize microcode load (IML) processes. While this was a strightforward operation, it was time consuming enough to delay a small number of I/O completions. Simplicity of solution was therefore graded medium.

4.2 Case Study 2: Broken Locks

Our second test was to force improper functionality of a cache software lock. The purpose of the lock is to protect meta-data related to the LRU replacement algorithm. Our fault injector simulated a rogue CPU that had gone into a loop in which it repeatedly took and held the cache lock. When this occurrs beyond an expected amount it prevents other CPUs from accessing that meta-data and can delay I/O completion. We tuned the severity of this fault by increasing the frequency and length of time the lock was taken over a 20 minute period (e.g. at low severity the lock would be taken once and held for a period of 50 microseconds, and at high severity the lock would be taken 10 times and each time held for a period of 5 seconds).

As we expected, availability was impacted by this fault. The longer the lock was held the greater the host impact. Beyond a particular point the host timed out on the I/O, which we recorded as a cascading error. We also found that, as in the first test, there was a recovery period during which performance was still impacted. Figure 3 shows the results from one test of high severity. In the graph, the lock was taken by the rogue CPU 3 times, and held for a duration of one minute at each instance. The

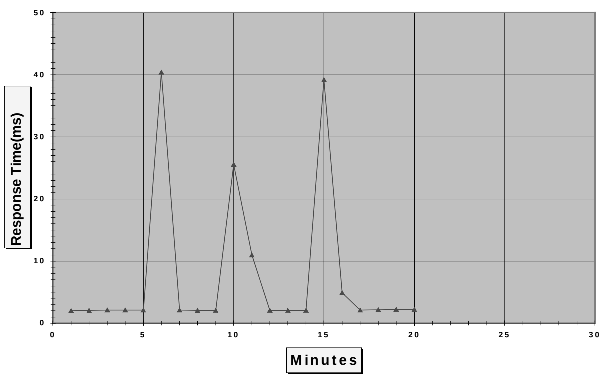


Figure 3: Impact on Response Time with Misused Lock (High Severity)

graphs shows a 30 minute period (represented on the x axis) versus the system's response time durring that period. At minutes 6, 10, and 15 the rogue CPU took the lock and held it for a period of 20 seconds.

At low severity (lock held for less then one second), the response time was not impacted. In between the two extremes of high and low severity, response time varied. We were able to gradually increase the severity to find where the effect began to become noticeable. Beyond a certain point a CPU waiting for the lock would assume the rogue CPU had malfunctioned and so would take the lock by force, thus the problem was self correcting. However, in our high severity tests the rogue CPU would continually re-aquire the lock.

We discovered several management utilities used the lock in order to function and their capabilities degraded when the fault was severe. This led us to grade robustness to be medium. We suggested a "force" option to these utilities to bypass taking the lock. In doing so, the acuracy of the utility would be impacted (because exclusive control of the meta-data would not be held). This would normally be an acceptable tradeoff, however, because in a real-time debugging scenario obtaining timely information is more important then perfect accuracy.

We found this problem difficult to diagnose. When we presented it to a customer service engineer (without telling him of our experiment), it took him a longer time to determine the root cause of the problem, relative to the first experiment. At the highest severity levels (an endless loop repeatedly taking the lock), the only course of action is to reset the CPU to terminate our fault injecting program. We graded simplicity of the solution to be low.

Because the available management utilities were relatively obscure and cryptic (they were used primarily by developers as opposed to system operators), we graded their effectiveness to be medium. We recommended adding a new counter into the lock mechanism in a prominent display utility. It would be displayed in a red if the lock was taken more than some number of times within an interval.

4.3 Conclusions from Case Studies 1 and 2

With the knowledge gained by these two tests we could more easily determine whether devoting development time towards early detection of these faults would be useful. In the first case, a complex mechanism to ensure synchronization between the CPUs could be designed. But because the complexity would be high and the impact of the bug was relatively low (and simple to detect were it to actually occur), the arguments to stick with a simpler recovery mechanism won out. The second problem (greedy locking) was more severe, but because of the unlikelyhood of its actual occurance we confined our recommendations to improvements to management utilities.

5. Case Study 3: Untargeted Fault Injection

In this section, we discuss our tests for simulating hardware faults in the cache memory. The goal of these "untargeted" tests was to verify that in the presence of these conditions the system would generate a brief sequence of transient errors before fencing off the offending memory, switching to a write-through mode (if not in one already), and alerting the operator.

A hardware fault manifests itself when the software fails to perform a read or write operation to the cache. Whenever this occurs, the software should follow a recovery path rather than assume the operation succeeded. Because the cache is accessed from many points in the system, there are many recovery paths to test. Rather than laboriously test each one (the targeted aproach), we created faults randomly and attempted to get a statistical sense of the correctness of the recovery paths.

The fault injection software works by inverting enough bits in a cache word to defeat the error correction code (ECC). On our system there is a mechanism to disable ECC generation. We disabled ECC generation, wrote random data into the cache word, then re-enabled ECC generation. At that point the ECC no longer corresponds to the data and so will flag an error when it is accessed. The location affected in the cache was adjustable to be LRU

Maintenance Tests Results			
Measurement	Case Study 1	Case Study 2	Case Study 3
Coverage	Targeted	Targeted	Untargeted
Diagnosis Effectiveness	Medium	Medium	High
Diagnosis Robustness	High	Medium	Medium
Solution Simplicity	Medium	Low	High

algorithm meta-data or overall system state. Although certain regions of memory (such as system timers) are more likely to be accessed quickly by a CPU (thereby triggering the fault) than others, all memory errors will eventually be found by background memory scrubbing processes.

The fault injection software's severity was adjusted by increasing the number of faults created. In our experiments, we increased the number of memory faults in granularities of 2, 10, 100, 200, 400, 1000, and 250000000. We did not benchmark availability using the performance techniques of the previous section because the act of fencing memory directly affects performance because the size of the cache shrinks. An availability benchmark would have to distinguish that expected degradation from the unexpected consequences of software errors. We left the project of learning those interrelations to future work.

At low severity (granularities of 1,2,10), our tests showed some diagnostic utilities functioned marginally slower, and a limited number of transient errors (affecting neither availability nor data) were generated. When we increased the severity to between 200 and 1000, the number of transient errors grew slightly, and we encountered errors in two out of 10 instances that did not recover immediately.

At high severity, we found some diagnostic utilities functioned very slowly when a large number of multi-bit errors were inserted. For example, one utility scans the cache to count the number of data blocks that are dirty, and in so doing, accesses the cache many times. If those accesses touched a "broken" cell in memory, the code would try again, repeatedly, until a timeout occurred. This is because accessing global memory was done through a wrapper function, which checks for errors and optionally retries the access some number of times if there was an error. We found in the case of the utility in question, the optional "retry mode" was enabled. That caused the utility to run so slowly as to be practically unusable. Our recommendation was to invoke the wrapper such that it would only attempt accessing the cache once.

In another case, we found diagnosis utilities that swept the cache would terminate after encountering the first memory error. This slowed the recovery process faced by customer service engineers. We recommended modifying the utility to continue processing the entire cache. Overall we graded robustness of tools to be of medium quality.

Diagnosis at each level of severity was straightforward: replace the "faulty" memory component. We had no difficulties executing that solution at any of the severity levels, and therefore we graded simplicity and efficiency of the solution to be of high quality.

An important lesson we learned was that correlating faults generated in this experiment to a particular piece of the software could be very difficult. For example, if 10000 faults are generated, and one of them causes a cascading error that impacts some other functionality, the problem becomes a tedious search for which of the faults has an incorrect recovery path that produced the problem. An important enhancement to the fault injector software will be to improve logging of where the fault was generated. In summary, we found that our diagnostic tools had several areas of improvement in the area of robustness.

6. Discussion

We found that rehearsing the corrective action with a customer service engineer was a useful technique. During such "fire drills" we could observe firsthand whether the system's diagnosis utilities were effective and how soon it would take to locate and fix the problem. We are skeptical that automated scripts could be devised to simulate real-time debugging, except under greatly simplified conditions.

An organization's QA department may be the logical home for reliability "regression" tests. Such tests would have to be carefully developed, because untargeted fault injection can create bugs that are difficult to find and may waste developer time. Conversely, targeted tests may be too specific and numerous to be efficiently used by a QA group already burdened with tracking ever-changing software. In many cases it may be preferable for reliability testing to be done according to the responsible developer's sense of the software's sensitivity to a problem.

We note there is another class of behavior that may negatively effect system availability. An “upgrade event” is some user-initiated modification of the system, such as hot-swapping a drive or portion of memory. For a continuously available system, such operations should minimize availability degradation.

We consider these to be a separate class of problems for three reasons: (1) these operations are normally part of the manufacturers regression tests and significant losses of availability should be detected at that time; (2) maintenance upgrade events are rare (hot-swapping a major component of the system typically happens on the order of months or years); (3) Availability degredation is expected as some upgrade events necessarily have some effect (e.g. such as hot-swaping memory can lead to cache misses), and many utilities (e.g. gathering exhaustive drive or cache statistics) executed durring the maintenance procedure will necessarily sidetrack the system from completing I/O requests.

It would be desirable to quantify the entire system’s reliability as a whole, rather than one component. Conceivably, this could allow two competing products to be compared against the same reliability benchmark. However, as noted by Prasad [10], this may not be possible. The range and variation of errors, and how they affect systems such as the one we test, is large. Separating a system such as ours into components (e.g. the cache) greatly simplifies the problem of reliability analysis [9].

We believe the problem of testing reliability of closed systems represents a fruitful area of research. It would be useful to obtain a sense of a system’s reliability using some agreed upon criteria or standard, as is possible with performance tests.

Achieving this goal will be challenging. Measuring the reliability of a closed system (e.g. comparing competing products) is difficult [12]. Closed systems limit the scope of fault injection because software fault injectors cannot be written and hardware data-paths are unknown. Additionally, because the system’s components (e.g. disk drives, memory cards), may be customized hardware, modifying them in a way to accurately generate a known fault is difficult. Finally, the management path (how system operators diagnose their systems in real-time) may be unobtainable.

Comparing the reliability of different storage systems is also complicated by the great variance of configuration options. Comparing “apples to apples” is very hard. To isolate testable similarities between competing machines, it is helpful to draw from experiences in performance testing. In such tests, certain subsystems play dominating roles, and other components are ignored. For example, in performance tests, the number and type of multiple input streams (SCSI, fast SCSI, fiber, ESCON, etc.), the number of drives, and the cache size are the most important variables that impact performance. The myriad of other potential configuration options provided (e.g. mirroring, RAID, dynamically attached disks), impact performance in a less obvious ways. We suggest evaluating two systems with the same configuration of dominant subsystems.

7. Related Research

Software fault injection (SFI) has frequently been used to generate untargeted faults. For example, by mutating code a programming error can be simulated [12]. Or, by simulating processor failure, a random event is created [2]. By repeatedly generating errors in this way statistics can be derived on reliability. Chen demonstrated that by using SFI, a positive development loop can be created to gradually improve the code [9].

The ISTORE project is studying reliability in storage systems, and has developed methodologies for availability benchmarking which we adapted for this paper. Their work has focused on examaning closed systems (e.g. Windows 2000) [1]. This work is directed torwards a development environment, in which any software fault instrument could be built for reliability measurement. Note that in development environment portability concerns (a frequent objection to SFI) are less of an issue.

Some work has been done to conceptually disassemble the complex software architecture that makes up a large storage system [6]. Kaaniche showed that once this is done, the problem of reliability measurement is somewhat simplified because lower “hierarchies” of software functionality may be viewed as a black box. Taking this useful perspective, our view in this paper was from the middle of the system, and the front and back end CPUs and devices were the “black boxes”.

8. Conclusion

In this report, we have described our experiences in applying reliability benchmarks to the cache subsystem of a large disk array. We successfully found deficiencies in diagnostic utilities and located the fault severity levels at which availability to the host became degraded. The work had a positive effect on the product as all of the deficiencies found were fed back to the designers who then made appropriate corrections. Two important lessons we learned are (1) testing techniques which focus both on particular algorithms and the overall system gave us a more complete picture of the system's reliability; and (2) maintainability is difficult to measure without human participation.

References

[1] Brown, A. Patterson D. "Towards Availability Benchmarks: A Case Study of Software RAID Systems." *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[2] Carreira, J. Silva, J. "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers." *IEEE Transactions on Software Engineering*. Vol 24, No 2. February 1998.

[3] EMC² Corporate Home Page: http://www.emc.com

[4] Kropp, N., Koopman, P. Siewiorek, D. "Automated Robstness Testing of Off-the-Shelf Software Components." *Fault Tolerant Computing Symposium*, pp. 230-239, June 23-25, 1998.

[5] Guedard, Y. Marneffe, L. Scheerens, F. Blanquart, H. Boyer, T. "Functional and Faulty Analysis: Some Experiments and Lessons Learned." *29th International Symposium on Fault-Tolerant Computing (FTCS-29)* Madison, Wisconsin, USA June 15-18, 1999.

[6] Kaaniche, M. Rmano, L. Kalbarczyk, Z. Iuer, R. Karcich, R. "A Hierarchial Appraoch for Dependability Analysis of a Commerical Cache-Based RAID Storage Architecture." *The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing.* 3 - 25 June, 1998.

[7] Koopman, P. "Toward a Scalable Method for Quantifying Aspects of Fault Tolerance, Software Assurance, and Computer Security." *Post Proceedings of the Computer Security, Dependability, and Assurance (CSDA'98),* 11-13 Nobember 1998.

[8] Michael, C. "On the Uniformity of Error Propagation in Software". *Proceedings of the 12th Annual Conference on Computer Assurance (COMPASS '97)*. Gaithersburg, MD. 1997.

[9] Ng, W. Chen, P. "The Systematic Improvement of Fault Tolerance in the Rio File Cache." *Proceedings of the 1999 Symposium on Fault-Tolerant Computing (FTCS)* , June 1999.

[10] Prasad, D. McDermid, J. "Dependability Evaluation using a Multi-Criteria Decision Analysis Procedure." *Proceedings of the Dependable Computing for Critical Applications.* January, 1999.

[11] Talagala, N. Patterson, D. "An Analysis of Error Behavior in a Large Storage System." *The 1999 IPPS Workshop on Fault Tolerance in Parallel and Distributed Systems.*

[12] Voas, Jeffrey. McGraw, Gary. "Software Fault Injection". Wiley Computer Publishing, New York, 1998.