Experience with OS Reliability Testing on the Exemplar System
How we built the CHO test from recycled materials

Danny R. Faught
faught@asqnet.org

Keywords: reliability testing, load testing, volume testing, stress testing, benchmarks, interactive simulation, parallel operating systems, supercomputers

Abstract

In this paper I will discuss the development and use of the Continuous Hours of Operation (CHO) test which was developed for the Convex Division of Hewlett-Packard. (The test was initially developed at Convex Computer Corporation before Convex was acquired by Hewlett-Packard.) My goal is to help readers with the practical aspects of developing a load test which can be used to help gauge and improve the reliability of a software system. I will briefly mention how we were unable to make much use of currently available software reliability frameworks, partially because of their deficiencies, and partially because of our inadequate knowledge of them. I knew enough about the theories to know that we did not have enough data to calculate a Mean Time to Failure metric, however, we were still successful in improving the reliability of our product and using the results of the CHO test as a release criterion.

Background

The System Software project leaders at Convex wanted an answer to the seemingly innocent question, "What is stress?" Since I was the leader of the team of test developers who were developing stress tests for the operating system, I was cordially invited to a project leader's meeting. I found out that what they really meant was, "What is a reliability test, and by the way, why don't you go ahead and build one?"

The System Software Test Group had a good foundation of functional tests, and we had a fairly comprehensive collection of volume tests which focused on particular areas of the system such as the file system, virtual memory, and networking. We were working on a category of "functional stress" tests that were even more focused. What we lacked, however, was a high-level reliability test that exercised most of the system at once.

The System Software organization's challenge was to measure and improve the reliability of the SPP-UX operating system running on a scalable parallel supercomputer, the Exemplar. Convex had experience with system testing our moderately parallel vector supercomputers, the C-series. We had a collection of user applications and volume tests that we would start simultaneously on a C-series machine and see what was left of the system a few days later. But the Exemplar architecture was different in many ways from anything we had worked with before, and the OS was incompatible with the C-series. So we started over, and we decided to try to mimic the end-user environment better.

The first Exemplar system that Convex released, the SPP 1000, supported two to eight PA-RISC CPUs in a single "hypernode," with memory sizes ranging from 256 megabytes to 2 gigabytes. Shortly thereafter, the 1000 was available in a multi-node configuration, with the architecture allowing for up to 16 hypernodes, for a total of 128 CPUs and 32 gigabytes of

memory. All hypernodes are accessible as a single system image. The Exemplar's NUMA (non-uniform memory access) architecture allows four different types of memory allocation, some local to a hypernode and some globally accessible across the system. The latest Exemplar models are the S-Class and X-Class, currently supporting 4-16 of the latest PA-RISC CPUs per hypernode, up to four gigabytes of memory per hypernode, and a maximum of four hypernodes in a system.

Other configurable features include "subcomplexes," which allow arbitrary groupings of the CPUs. Processes always run in a particular subcomplex, and access to subcomplexes can be selectively restricted. A subcomplex can span all hypernodes, and a multi-threaded process can use all CPUs within its subcomplex at the same time. There is a user-sizable cache for global memory, and the size of the global memory pool can be adjusted for each subcomplex. See [WWW], [HP96], and [HP97] for sources of additional information about the Exemplar system.

All this scalability and configurability means there are far more hardware and software configurations than we can test. Also, the sheer size and horsepower of the system made it difficult to build a test that could keep it busy. Developing the CHO test was a challenge because the Exemplar technology was breaking new ground. It took some experimentation to see what was required to exercise the various parts of the system. At times I could almost hear it laughing at me, but I got the last laugh, watching the system crumple into a pathetic pile of molten bits. :-)

Terminology

CHO is the name of both our reliability metric and the test we built to measure it. I refer to the "CHO test" as a shorthand for "CHO reliability test," though CHO is at a higher conceptual level than the tests and other programs it incorporates. We borrowed the "Continuous Hours of Operation" term from another Hewlett-Packard division which has been developing its own set of CHO tests independently since the mid-1980's. Further research will be required to determine the origin of the term.

Did we really do reliability testing? We had reliability goals, and we built and ran a test to help us measure and achieve those goals. However, as you read further, you'll see that we had to use gut feelings more than scientific rigor. If your definition of reliability testing requires this rigor, then CHO does not fall within the definition. So be it. I use a broader definition for the purposes of this paper.

Like most testing terms, the terms associated with system testing are loosely defined. At Convex, the component parts of CHO are commonly called "stress tests." However, stress testing is usually associated with exhausting a resource for a short period of time. The tests we used for CHO were designed to exercise the system over an extended period of time, while not quite exhausting all available resources. The most appropriate term to describe these tests is "volume tests." In fact, the tests would have to be made far more robust in order to behave in a sane fashion if one of them could steal all of a system resource from the others.

As documented in [KANER], Cem Kaner places both volume tests and stress tests in the larger category of "load testing," which tests "every limit on the program's behavior that any of the product's documents asserts," (page 134). He says volume tests "study the largest tasks the program can deal with," and stress tests "study the program's response to peak bursts of activity," (page 55). Rodney Wilson, in [WILSON], recognizes that many people have varying definitions of these terms, though he does not try to nail them down (pages 46, 82, 86). Wilson treats volume testing and load testing as synonyms.

The CHO test's aim is to keep many parts of the system very busy for an extended period of time. While there are some variations in the load, there are no resource-exhausting peaks that would characterize CHO as a stress test. Though both authors mentioned above state that volume testing pushes the system to its limits, it is also very important to exercise the system in a way that is more typical of what the customers will do with it. Here I use an expanded definition of volume testing that includes loads that are somewhat below the maximum that the system can handle, but sticks with the spirit of volume testing, in that the load must be applied steadily over an extended period of time. You do not have to hit the limits to find plenty of bugs! In fact, just below the limits you are likely to shake out bugs that will affect a broader base of customers. Of course, you still want to test those limits, but probably in the context of a brief stress test rather than an extended volume test.

Tests Used for CHO version 1

The beauty of CHO is its recycled content. Most of the harness development was already done, and we were able to obtain all of the tests from existing code. The CHO version 1 subtests are:

shell_stress - The main source of CHO's load. Shell_stress is a home-grown test that randomly selects shell command sequences from a database and feeds them to a shell. By using a combination of perl code and the "expect" tool from NIST, we created an environment virtually identical to that used by real users (see [PERL] and [EXPECT]). We enhanced shell_stress for CHO to provide networking coverage. Each simulated user does a remote login, since in real use the console is the only login session that is not a remote login. Also, a third of the sessions work in an NFS-mounted directory, in order to stress the NFS filesystem driver. All networking connections are set up in a software loopback to the same machine, so the test does not require additional lab resources for remote networking connections.

SPECint - A well-known benchmark that runs several programs that mimic various customer loads.

a *customer application* - A proprietary number-crunching application provided by a customer. The application was known to find reliability problems in the operating system.

vmstress - A home-grown virtual memory test.

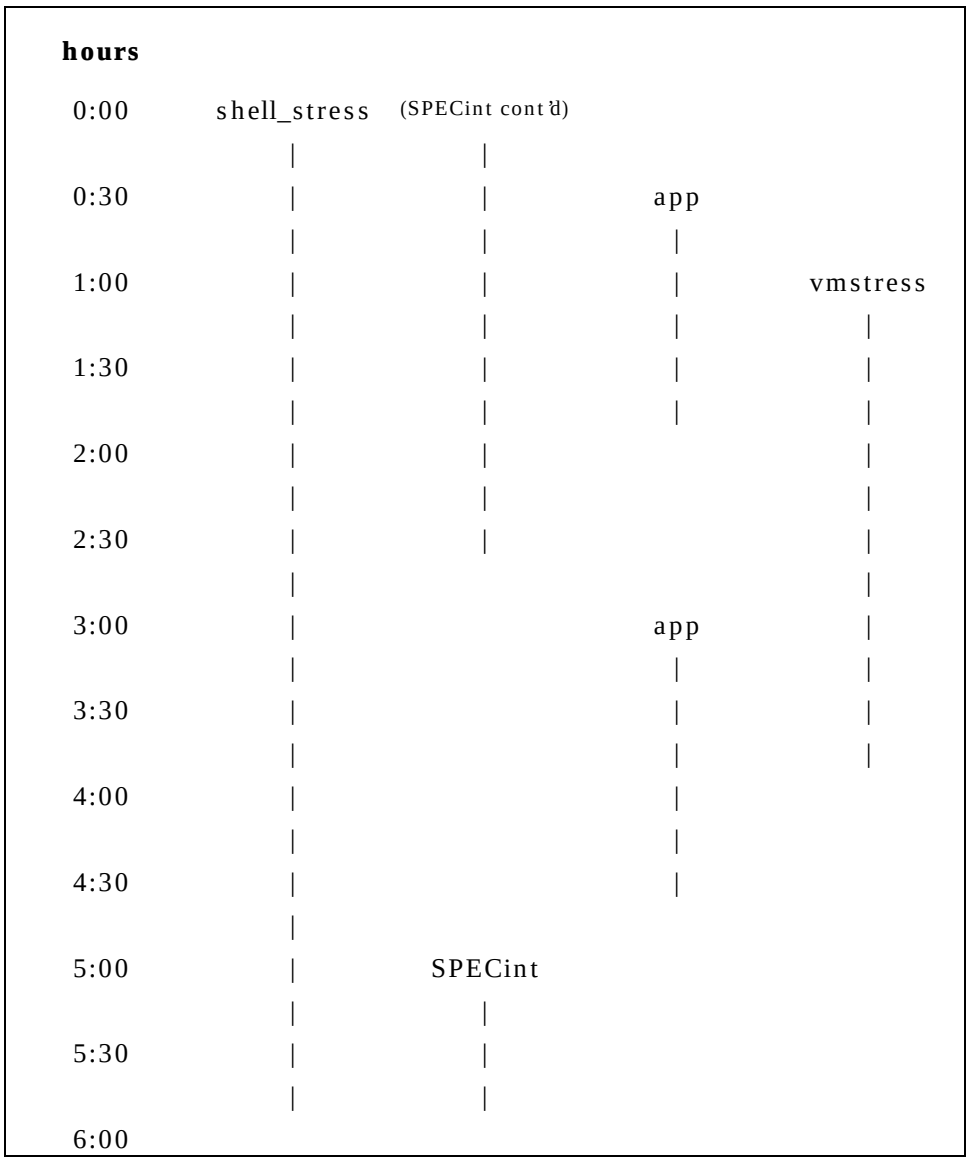
We consulted with customers to see what their expectations were regarding the sustained load level that the system should be able to handle, and we made sure the CHO test exceeded these loads. But we have not yet attempted to use statistical sampling to generate a rigorous operational profile. Different customers put very different kinds of loads on the machine. I have heard of system call usage samples which even varied widely on the same machine from one day to the next. We put our database together based on some informal internal sampling of utilities, and we limited ourselves to the programs which were readily available on the system. Obtaining a broad range of applications which the customers use on our systems would be very expensive and time-consuming. We will continue to look for a feasible way to put together a more representative sampling for our database.

One technique which we have used recently is to design a test which mimics the load from one of the data management applications that we offer. The test is less complex than the application itself, and it will hopefully prove to make debugging much easier. There are no current plans to include this new test in the CHO framework, though we may consider this if the test is successful.

To simulate a customer environment better, we set up a varied schedule for running the tests rather than running them all continuously, as shown in Table 1. We could have allowed shell_stress's scheduler drive all of the tests in a random fashion, in fact, this simplistic

approach seems to be the standard way to run a reliability test. But we wanted to control the execution of the larger applications, since customers do not blindly start large jobs on an already heavily-loaded system. It would have been much more difficult to allocate the system's resources if it was possible for several large jobs to be randomly scheduled at the same time.

Table 1. CHO version 1 test schedule



Shell_stress runs continuously, though its intermediate scheduler varies the load somewhat. The customer application starts 30 minutes after the beginning of the run and runs 1 hour 30 minutes. Then three hours into the run, it is started again for an hour and 45 minutes. The vmstress test starts an hour into the run and runs for three hours. To make the schedule seamless, SPECint is set to start five hours into the run and continue for 3 hours 45 minutes, which wraps around the beginning of the schedule. That means at the beginning of the run, the harness needs to kick off an abbreviated 2 hour 45 minutes run of SPECint. All SPECint runs after the first in a session use the full time slot.

The schedule was designed carefully so that the cross-section represents a varying load. There is one 15-minute period where only one test is running, and an hour where all four tests are running. All other combinations appear on the schedule, assuming the constraint that shell_stress is always running.

The user application and SPECint were not designed to run indefinitely, so they are run in a loop during their time slots. They are killed with a signal at the end of their time. Vmstress runs continuously until the harness signals it at the end of its run, and shell_stress runs continuously unless the number of errors passes a defined threshold, in which case it exits and the harness restarts it immediately.

Tests That Did Not Make the Cut

We experimented with a multi-user benchmark as a part of CHO, and when we set it to simulate a large number of users it did shake out some OS bugs. However, it did not seem to be well-tested at such high load levels; we ran into a number of bugs in the benchmark itself. We chose to use SPECint instead.

We also had originally wanted to include a disk I/O test in CHO. However, there was an I/O bottleneck in early versions of the OS. We could be running the rest of the tests and achieving a load average of 100, but when we started the disk test, the load would drop to 5. The tests seemed to be piling up on contentions for disk access. We submitted a bug report on the I/O performance and yanked the test because the defect would not be fixed until after the upcoming beta release. I prefer to generate disk loads as a natural consequence of running a customer application, anyway.

We also wanted to test subcomplex reconfiguration. We anticipated that customers might reconfigure the subcomplexes on the system after most interactive users go home, to give the system more horsepower for batch runs. However, subcomplex support in the OS was immature at the time, and subcomplex handling in the harness was poorly tested. We decided to reduce the complexity of CHO by using only the default subcomplex.

There were several other situations where the OS behaved badly when running CHO. However, we usually insisted that since it was the OS that needed fixing, and not the test, we would not change the test to work around the problem. It was important that CHO remain as static as possible after we started using it for a particular release, so we could compare the results from different runs. However, in one case shells run by shell_stress hung in the exit path and tied up system resources. We had to implement a reaper to kill these hung shells. During each release cycle we checked the reaper's log to verify that the bug was still there. The bug was not given a high priority because it had not been reproduced in actual use on a timeshare machine, but eventually it was fixed and we removed the reaper.

The Sweat Harness

When we were designing the CHO test, the test group had already developed the Sweat test harness for the purposes of burn-in testing on the manufacturing floor. We felt that a few enhancements could make Sweat a suitable harness for CHO. We recruited one of the original Sweat developers and explained what we needed. It took a good deal of scribbling on the whiteboard to show how to integrate the new scheduling features and how to handle non-obvious subtleties such as starting abbreviated overlapping test runs when the harness starts. After we hammered out the design, the necessary code changes were not too drastic.

The tests that Sweat runs are specified by a sequence file. The essential elements of the sequence file used by CHO version 1 are shown in Table 2.

Table 2. Sequence file for CHO version 1

Test	Timeout,How	Delay
shell_stress	0,pass	0
App	360,pass	120
App	420,pass	720
Vmstress	720,pass	240
SPECint	900,pass	1200

The test name matches a directory within the Sweat test suite. When it is time to run the test, Sweat executes a file called "ExecTest" in that directory. This file is typically a shell script or perl script that performs the run-time setup and executes the test program. This allows a great of deal of flexibility in running the test and still keeps the interface very simple. It is also convenient to know which file to look at first in a test directory when you want to see how everything gets started.

The duration of each test is controlled by the timeout field, indicated in minutes. We have found that it is easier to let the harness control the test run durations rather than having each test implement a timeout mechanism. The ExecTest script for each test is designed to catch the timeout signal and then clean up and exit. If they did not catch the signal, they would die from the signal and Sweat would log a failure. The "how" field for each test is set to "pass". This tells Sweat that a timeout is a normal occurrence, so it should not indicate a failure when the timeout arrives before the test exits.

The delay field tells Sweat how long to wait after the start of a cycle, in minutes, before starting the test. This is what allowed us to stagger the start of the tests in order to vary the load. Note that the user application is listed twice in Table 2, with different timeouts and delay times. This results in two different runs of the test in different time slots.

When we run Sweat to start CHO, we tell it to repeat the sequence indefinitely. Sweat assumes that the time frames in the sequence file are based on a 24-hour period. We wanted to repeat the sequence more frequently than that, so we used Sweat's scale option to scale the sequence down to 25%, or 6 hours. So the timeout and delay numbers in the sequence file are divided by 4. It is important to get through all the various phases of the schedule fairly quickly because the most blatant functionally-oriented bugs are found by the end of the first pass. This is especially important early in a release cycle when reliability is still relatively low.

We did not support an off switch for the test, though we did attempt to get the test to abort as cleanly as possible when the front end was killed. To do this, we tried to allow for signal propagation down the process hierarchy, which could get 10 levels deep at some points. It was difficult to assure proper signal propagation at each level, especially when we could not modify some of the programs. The Bourne shell was especially problematic with respect to signal handling. We were able to alleviate the problem in some cases by using "exec" to start a new program without involving another process. This is similar to tail-recursive optimization– if you no longer need the parent process to hang around waiting for the child, then do not leave it there. When we do an exec, any signals go directly to the new program. In other cases we solved the problem by converting shell scripts to perl in order to use perl's signal handling features.

The run ends when the system crashes or becomes comatose. Sometimes a system failure is easy to identify, like a nice panic message on the console. Often, however, parts of the system

simply hang. Sometimes, the system even recovers from a brief hang, making it difficult to detect the problem. Sweat does not attempt to deal with these issues; it simply plows ahead as long as it can. We can abort the test by rebooting the system. In fact, rebooting while under a heavy load is a wonderful test of all the shutdown mechanisms.

The actual program we use to start the CHO test is a script named "run-cho." It takes care of the setup that cannot easily be done within the Sweat framework. If any test is unable to run, the results of the CHO run are invalid. Since an individual test cannot cause the CHO run to abort, the run-cho script gives each test an opportunity to abort the run before Sweat is invoked. Run-cho does this by running a program named "Verify" in each test directory. If the Verify script returns a nonzero value, the run is aborted. Run-cho takes care of other housekeeping, such as setting up a log file and establishing a heartbeat that runs the uptime utility every 5 minutes. Examining the uptime data in the logs can provide valuable data. For instance, if the run was unattended, it establishes the time to the nearest 5 minutes that the system crashed. It also shows a history of system load and simulated users logged in, which will fall too low if the system starts to malfunction before it completely halts.

The relationships among run-cho, Sweat, and the tests follow what I call the "sandwich model." This model represents a pattern I have noticed in several test environments. We start with a generic harness, Sweat in this case, and we build our particular tests into the harness's framework. Then we realize that the harness is deficient in some way, and we do not have the time or authority to enhance the harness, so we build a wrapper on top of it. We end up with the generic harness sandwiched between our environment-specific software for the front end and the underlying tests.

Though this model has provided a practical way to solve problems, it is not ideal. We would prefer to have the same user interface for all test environments that use the same harness rather than a different customized front end for each. One way to solve the problem would be to provide hooks in the harness to make it more flexible, so we can program the harness to handle all of the setup. Then users could execute the harness directly. For example, the Verify functionality in run-cho really should be part of the Sweat harness. However, because Sweat is currently used in only the one test environment (its use in the manufacturing area is no longer supported), the return on that investment would be minimal. In the case of our functional test harness, the user base is much broader, and we have successfully moved functionality from our local test environment into the generic toolset. However, it does not appear that all the features of our front end will ever be incorporated into the common harness, because the front end has grown to be quite a large beast, with many features that might hamper other groups' use of the harness. So we are stuck with the sandwich model.

Another way of addressing the need for a common user interface is to use a generic front end that can be customized for each environment and all harnesses and yet provide a consistent user interface. We attempted this approach, but at that time the project proved to be more ambitious than we could handle.

Scaling

The CHO test is scalable in a couple of different ways. Scaling based on the configuration is automatic, and the user can control an additional scaling factor by setting the "stress_factor" environment variable. This variable ranges from 1 to 10. A stress_factor of 1 is the lowest interesting load, and it is the value we usually use when trying to get an "official" run. A stress_factor of 10 is intended to be the highest practical load that can be applied to the system. Any integer in between can be chosen. Users of the test do sometimes use higher stress_factor values in order to accelerate the detection of bugs.

The user application and SPECint are not set up to scale. The vmstress test does scale, with the number of memory pages used calculated like so:

(free_memory_pages - 1000) * (adjusted_stress_factor / 10)

We calculate the number of total free virtual memory pages each time vmstress runs. 1000 pages are reserved so the basic system services and the harness can continue to function (since the objective is volume rather than stress). We calculate the adjusted_stress_factor by adding 3 to the stress_factor and truncating it at 10. We make the adjustment because we judged that the test was not stressful enough in this environment at the lower stress levels. By adjusting the stress_factor, we are able to leave the test code alone and simplify the maintenance of the test under the two harnesses that use it. The adjusted stress_factor is divided by 10 to give a percentage of available memory to use (40-100%).

In hindsight, we would rather have based the calculation on the amount of free swap space instead of total free virtual memory, because on our large memory systems the vmstress test often did not cause any paging at all because the ratio of physical memory to swap space was high.

The shell_stress test scales the number of users like so:

maximum = cpus * adjusted_stress_factor
minimum = maximum * .75

Shell_stress uses an intermediate driver that starts out with the maximum number of users and varies it between the maximum and 3/4 of the maximum. We adjust the stress_factor here the same way we do for vmstress.

CHO version 2

After the CHO test had stabilized somewhat, we decided to correct some of the deficiencies of the design so that the test would do a better job of exercising the features of SPP-UX that our customers use. Shell_stress was kept pretty much as is, though it had been enhanced throughout the development of version 1. Vmstress was replaced with "gsmstress," a home-grown test that specifically tests SPP-UX's global memory. We replaced SPECint with the Naspar parallel benchmark in order to add some load from multi-threaded programs. The proprietary user application was replaced with a Gaussian92 test. Gaussian92 is a third-party application that is process-parallel and is used by a broader user base than the application it replaced. We added "pong," which tests message-passing, a popular programming model that is an alternative to using processes that span multiple hypernodes and communicate with global memory. And we added "ftpstress," a home-grown networking test ported from our C-series tests that exercises the ftp client and server over a loopback. Ftp is an underlying protocol used on many of our file server systems.

We also had a request to add a test that exercised the magnetic tape device and drivers. However, because this would have required physical access to the system, it was determined that this was not practical. Administration of the CHO test and many aspects of the administration of the Exemplar system could be managed remotely, even removing and restoring power. Engineers could start and monitor a CHO run from home, which was especially important when a run stretched into a night or over a weekend. So having to insert a tape would not be advantageous. Also, requiring a tape device would have limited the choice of lab machines, and we would have run the risk of a false test failure because of worn out tape media. So we reluctantly decided to keep the tape stress tests separate from CHO.

Another enhancement we added was taking better advantage of subcomplexes. CHO version 2 allocates half of the available CPUs to the default subcomplex, named "System." A quarter of

the CPUs go to the "Stress" subcomplex and a quarter to the "Appl" subcomplex. We still have not attempted to dynamically reconfigure the subcomplexes during the middle of a CHO run. The schedule and subcomplex allocations are shown in Table 3.

Table 3. CHO version 2 test schedule

Subcomplex:	**System**	**System**	**Stress**	**Appl**
Hours				
0:00	shell_stress		gsmstress	Gaussian92
0:30		pong		
1:00				
1:30				Naspar
2:00				
2:30				
3:00			ftpstress	
3:30		pong		
4:00				
4:30				Gaussian92
5:00				
5:30				
6:00				

This schedule is set up under Sweat with the configuration information shown in Table 4.

Table 4. Sequence file for CHO version 2

Test	Subcomplex	Timeout,How	Delay
shell_stress	System	0,pass	0
pong	System	480,pass	120
pong	System	480,pass	840
gsmstress	Stress	600,pass	0
ftpstress	Stress	600,pass	720
g92	Appl	240,pass	0
g92	Appl	240,pass	1080
Naspar	Appl	600,pass	360

We were concerned about the fact that the makeup of the new version of the test was so different from the first version that comparisons with past results would be meaningless. For a while, we ran both CHO versions on different machines both running the latest OS. We saw that CHO version 2 actually generated a lower load average on the system, but load average is often not a good indicator of the stress on the system. The newer test put a higher demand on system resources, thus making less time available for user processes, which is exactly what we wanted.

We performed an experiment to see how the previous release of the OS performed under the new CHO test. The system did not fare well at all, much worse than with the first CHO. Part of this may have been because the new test was more stressful overall, but I think the real factor is that this was the first time we had applied that particular mix of tests to that version of the OS. The first version of the test had already run into the "pesticide paradox" ([BEIZER], page 9), since the OS was basically trained to deal with it. CHO version 2 broke us out of the paradox, giving the OS a new challenge.

Revision Control

Revision control proved to be difficult for CHO. Some parts of it were already in our revision control system in the format used by our functional test harness. We did not want to copy these files into a different section of the revision control system for CHO. At one point, my work tree contained links to our main release tree so we would always pick up the latest versions when we released CHO. However, we decided that we needed more direct control over new revisions that were picked up, so we handled the updates manually. In lieu of proper revision control, we kept copies of each CHO release, each clearly labelled. We kept several recent revisions on disk, and older revisions were on tape due to the large space requirements (greater than 70 megabytes per release, mostly due to SPECint). Only once did we need to access the tape to check an old revision. We documented incremental changes in a release notes file which we maintained as a part of the CHO product.

My group has now converted its revision control system to ClearCase. It is likely that we will be able to use ClearCase to provide revision control for CHO since ClearCase directly supports links.

Bootstrapping

Developing the CHO test was a very long, incremental process. The very first run lasted about 2 seconds before the system crashed! While the test had obviously succeeded in finding an operating system bug, the crash masked a large number of latent test bugs. It wasn't as bad as it could have been, since the individual parts of CHO had been run in isolation. Most test bugs were integration problems in the glue that held all the tests together.

I picture the maturation of any of our system tests as a process of pulling up the test and the software under test by their bootstraps– first one gets a little better, revealing a problem in the other, then the opposite. We go back and forth, fixing OS bugs and test bugs. As the OS gets better able to run the test, the test gets better able to exercise the system. It was initially impossible to deliver a well-tested test product because there was no stable system on which to test the test. Unfortunately, at the time, the test group was not doing formal inspections or code reviews, so we relied heavily on testing quality into the tests. We know better now, but we are still faced with the problem of doing final testing of a test product on an early version of the OS.

Because many executables can run on both SPP-UX and HP-UX, we did have somewhat of a reference platform. But SPP-UX includes a number of enhancements beyond HP-UX, and some of the CHO tests necessarily exercise those enhancements. We were able to qualify some parts of CHO, such as shell_stress, using HP-UX. But the workstations we could run HP-UX on were much smaller than the Exemplar systems (fewer CPUs, less memory, etc.), so we could not run the tests at the higher load levels that would be used for SPP-UX.

As we continue to use the CHO test, the OS continues to improve. There is a higher likelihood of getting a long test run, and the OS performs each step better along the way. The remaining test bugs are mostly subtle problems that are no longer masked by the more serious bugs. Also remaining are some more serious problems that are very difficult to isolate (we don't know whether to blame them on the test or the OS), but we have added instrumentation to the test that helps to isolate them. As the rest of the system matures, it becomes easier to track down the cause of the problems.

Reproducibility

Ideally, when you get one run of the CHO test, it would run for exactly the same length of time if we run the test the same way again. Then we would know exactly what the Mean Time To Failure metric is with only one run. But most OS bugs that affect stability are nondeterministic, especially on a complex parallel system. A "rolling the dice" metaphor is the best way to think about them. A bug may be a timing problem that has a failure probability of one in a billion. Each time we roll the dice through that code, possibly thousands of times per second, we may encounter a failure. It may happen on the first roll, or after five billion rolls. That is why it is important to consider the "mean" in "mean time to failure."

Nondeterministic bugs easily explain why our results varied so much despite our best efforts, especially when we consider that there can be several such intermittent problems latent in the system. For example, one string of results gave these run times on the same OS: 14 hours, 1 hour, then 43 hours. While I am no statistician, intuitively I would put very little confidence in the average of these numbers.

Another factor is that we had two classes of results– runs that ended with a system failure, and runs that ended artificially. The most common reason that runs were artificially aborted was because the reliability goal was met. So we ended up with a successful run, but the number represented a run time that was merely a lower bound of the true capability of the system. It does not make sense to average these aborted results with results that did end with

a system failure. This also causes more uncertainty when we get a few runs that are aborted when they meet the goal, and a few runs that crash before meeting the goal. Perhaps the actual MTTF of the system far exceeds the goal, but if we naively average all the numbers, we incorrectly get a number smaller than the goal.

We made a futile effort to improve reproducibility by using the same random seed for each run (shell_stress uses random numbers). Ideally, this would result in the exact same random numbers and random decisions being used each time. However, there is a nondeterminism in the order that the OS completes its tasks, so the random decisions would sooner or later be applied in different places than the last run, and the run would diverge from there.

We also put restrictions in place that were designed to aid in reproducibility. We required that the system be rebooted no more than 30 minutes before starting the run. We asked that the person monitoring the run login only on the console and not start any remote logins. We asked that the engineers keep their activity on the system to a bare minimum. This was more important early on when the system was unstable. Just running a monitoring tool could have a dramatic effect on the results. Later on, as the system became more stable, we tended to use monitoring tools more because their effect on the system was less pronounced, and the data they provided was more important than the reproducibility of the results.

Even with these restrictions, there is a good deal of variation in the paths that the OS traverses before and during a CHO run as the engineer monitors the progress. But no matter how strong the restrictions are, even to the point of completely automating the run, it is still impossible to ensure that the same paths are exercised every time. For example, variances in peripheral timing during bootup may cause subtle differences in the OS's response.

We theoretically could still get enough statistical samples such that the result stabilizes. However, it is practically impossible to get enough samples, and as cycle times for new releases continue to get smaller, the number of samples we can get will continue to shrink. We usually got 1-3 runs on a particular build of the OS before applying bug fixes and moving on to the next build. This is not nearly enough data to be able to draw any mathematical conclusions. In [LYU], Brocklehurst and Littlewood report "...the length of time needed for a reliability growth model to acquire the evidence that a particular target mean time to failure has been achieved will be *many times* that target," (page 159).

Results

While we wanted to use the CHO test to measure the reliability of the system, it was also very important that we improve the reliability. CHO could be used as a load test when we wanted to find bugs as efficiently as possible and we did not need to get an official measurement. In these cases, we would relax the preconditions and crank up the stress factor. My hypothesis is that the set of bugs we find while using a stress factor higher than the default overlaps the set of bugs that affect the measurement runs, though I have not determined how large the overlap is. Using CHO as a load test is helpful, but it does not always give us forward progress on our measured reliability at the minimum stress factor.

The development of the CHO test started in 1994, and it started to be used seriously as an OS release criterion in 1995. We continue to use the CHO test now in 1997 to shake out bugs and to help measure the reliability of the system. At one point along the way, we realized that we were relying too much on the CHO test, so we started paying more attention to other system tests which we could run at higher load levels when run in isolation. It is unwise to rely on one metric to determine the reliability of the system. There is too much risk that one test will miss a problem that impacts the customers.

Since we do not get enough data to calculate a Mean Time to Failure metric, we generally rely on a gut feeling to decide when we have met our reliability goals. We like to see an upward trend in the CHO run time as each new OS build is produced. We like to see two to three runs on the same build that meet the goal, on at least two different hardware configurations. If we also get shorter runs, the severity of the problem is evaluated to determine whether to hold off the release.

One difficulty we have to deal with is system crashes that occur long after the system starts to exhibit symptoms. It is fairly common for a system malfunction to occur that does not completely halt the system. There have even been cases where the uptime monitor was practically the only part of the system still running. So it can be misleading to rely on the last time stamp given in the log file before the system crashed. Engineers would examine the data from the whole run, graphing the load average and number of simulated users logged in over time. If there was a downward trend or a discontinuity long before the system crashed or the run was halted, they would report the time the problems first occurred. Unfortunately, this process is subjective and it relies on the experience of the engineer.

We found that a CHO run seemed to build up momentum after a while, so if it ran a significant fraction of the goal time, it was likely to meet and far exceed the goal. Systems in the field have run into bugs that take weeks to reproduce, longer than we typically can run in the lab. But often we were surprised with a very short run. This seems to suggest a miniature "bathtub curve" for our software. This curve, as discussed in [WALKER] and elsewhere, is a common phenomenon for hardware, which tends to have many failures when first installed, then perform well for a long time until parts eventually start to wear out. Perhaps in a limited sense it also applies to software during a single boot of the system.

We have not attempted to predict field reliability using the CHO results, because even the least stressful CHO run puts more load on the system than the average machine in the field encounters (the load average on a system running CHO is normally in the range of 20-150, depending on the size of the system). We have not yet tried to calculate a multiplier that would correlate our CHO results with the field reliability, and I am not sure that it makes sense to do so before we are capable of computing an MTTF metric. I think my theory about an overlapping set of bugs found by high and low CHO loads can be extrapolated to say that the bugs found by CHO with a stress factor of 1 finds a set of bugs overlapping those bugs seen by customers at more reasonable loads. Because the cost of our lab machines is measured in hundreds of thousands of dollars each, we did not have the luxury of running CHO at a lower load level for a much more extended period of time.

The operating system has improved dramatically after that first two-second run on an alpha version of the OS. Run times increased to hours and then days, and it is common to abort a run the when goal has been exceeded and the lab machine is scheduled for other activities. There is no doubt that we have greatly increased the reliability of the system. In many cases the CHO test found problems that were not found by our other testing, and in other cases it found them earlier than we otherwise would have found them. We will continue to learn from our practical experience with CHO, and we will continue to learn about the established reliability theories and how they might can help us conduct more accurate reliability assessments.

Acknowledgments

While I led the initial development of Convex's CHO test, I have since passed the development and maintenance on to other capable hands. My thanks go to Toru Iino and Aaron Haney for adding their own enhancements and fixing many of the bugs I left in it. Also, thanks go to Daryl Wray for encouraging me during the initial development of CHO, and all the OS

developers who have put CHO to good use. And I thank Boris Beizer for encouraging me, in his own unique way, to write about testing.

References

[BEIZER] Boris Beizer, *Software Testing Techniques*, second edition. New York: Van Nostrand Reinhold, 1990. ISBN 0-442-20672-0, 1-85032-880-3.

[EXPECT] Don Libes. *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs*. Sebastopol, CA: O'Reilly & Associates, Inc., 1995. ISBN 1-56592-090-2. See also http://expect.nist.gov/.

[HP96] *Exemplar SPP1000-Series Architecture*, fourth edition. Richardson, TX: Hewlett-Packard Convex Division, 1996, Order No. DHW-014.

[HP97] *Exemplar Architecture: S-Class and X-Class Servers*, first edition. Richardson, TX: Hewlett-Packard Convex Division, 1997, Order No. A4716-90001.

[KANER] Cem Kaner, J. Falk, H. Q. Nguyen. *Testing Computer Software*, second edition. New York: Van Nostrand Reinhold, 1993. ISBN 0-442-01361-2, 1-85032-847-1.

[LYU] Michael R. Lyu, ed. *Handbook of Software Reliability Engineering*. New York: McGraw Hill, 1996. ISBN 0070394008.

[PERL] Larry Wall, Tom Christiansen, Randal L. Schwartz. *Programming Perl*, second edition. Sebastopol, CA: O'Reilly & Associates, Inc.: 1996. ISBN 1-56592-149-6. See also http://www.perl.com/perl/.

[WALKER] Ellen Walker. "Software/Hardware Reliability - Bridging the Communication Gap." *RAC Journal*, Reliability Analysis Center, 2nd Quarter 1996, Vol. 4, No. 2. Text available at http://rome.iitri.com/RAC/DATA/JOURNAL/2ND_Q1996/software_hardware.html.

[WILSON] Rodney Wilson. *UNIX Test Tools and Benchmarks*. Upper Saddle River, NJ: Prentice Hall, 1995. ISBN 0-13-125634-3.

[WWW] For more information about the Exemplar system, see the web page http://www.convex.com/.