# The Making of an Open Source Stress Test Tool

Danny R. Faught
http://www.tejasconsulting.com
faught@tejasconsulting.com


The objectives of the paper are:

1. Introduce you to an open source stress test tool, the stress_driver, which you can use and modify in your own environment.
2. Explain the benefits and limitations of a general-purpose reusable stress test tool.
3. Describe some of the implementation details involved in building a heavy-duty test driver using a scripting language.

## History

The saga of the stress_driver tool starts in 1993, when I wrote a prototype of a general-purpose stress test tool using the Perl scripting language. I then handed it over to another test developer to reimplement in a compiled language because I felt that a Perl script would not have sufficient performance to be able to stress the supercomputers I was testing.

A year later, we were trying to track down some mysterious problems in the tool, and I declared the C++ version to be unmaintainable. It didn't help that the programmer had left the company and we didn't have many C++ experts on staff. So I dusted off the Perl version, and that's the code base that survives to this day.

Some very recent news that makes the story much more interesting is that Hewlett-Packard, the current owner of the stress_driver tool after acquiring Convex, has granted an open source license for a large body of test tools and automated test cases, including the stress_driver and a suite of stress tests that use it. So I (no longer an HP employee) have been able to resume the development of the tool.

The code is now available for download, but it's buried within 21 megabytes of other data, nobody knows that it's there, and it would only work on specially configured versions of Perl running on SPP-UX or HP-UX 9, which few people now have access to. I have ported the stress_driver to Windows and Linux using the standard Perl distribution so that it's useful for a much broader audience.

The stress_driver tool is currently about 700 lines of Perl code, plus a manual page. The stress_driver runs a given test program, possibly scheduling random numbers of parallel invocations and randomly choosing parameters based on the user's

specifications. It can scale the load based on the number of CPUs in the system and a user-specified 1-10 scale, and it can run the tests under different user IDs. It can vary the run time of each test process within a given range, and it can vary the number of parallel invocations of the test within a given range using an adaptive scheduling algorithm. The tool was used by a suite of operating system stress tests, testing the filesystem, memory management, process and thread management, and it was also a key part of a large-scale system reliability test.

## What stress_driver does

The stress_driver is a generic stress test tool; you must provide a test program for stress_driver to run. You can use stress_driver in a variety of ways. There are basically two different ways it interacts with the test program. If the test program is designed to run for an indefinite period of time, then stress_driver will run the program once for each time slot that it sets up for the test, and it will kill the test at the end of the time slot. The degenerative case is when you only want one copy of the test to run at a time. Stress_driver doesn't add much value in that case, except to stop the test when the time period that you specify is done.

You can tell stress_driver how many tests to run at a time, and stress_driver will start that many copies of the test. You might randomize the parameters that each test receives, and you might scale both the test's parameters and the number of test programs according to a 1-10 scale provided at runtime. Stress_driver also used to be able to scale up automatically based on the number of processors on the system, but I've disabled this feature until I get access to another multiple-processor system.

The second type of interaction with the test program is the case where the test executes some defined transaction and then exits. In this case, stress_driver will usually need to schedule more than one iteration of the test during each time slot. Perhaps the degenerative case of just running one test at a time is somewhat more useful here, because stress_driver will continue re-running the test until the specified time period or number of iterations is complete.

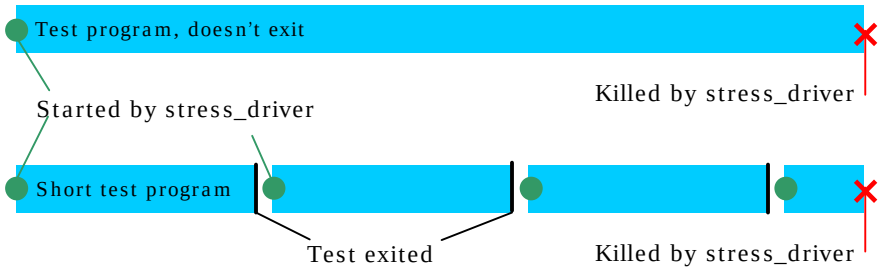


Figure 1. One stress_driver time slot, for a test with no built-in end point, and for a short-running test.

Here's an overview of stress_driver's options. For a more complete reference, see the manual page in the stress_driver source distribution. A Unix man-style synopsis is:

stress_driver [–log *logfile*] [–life *time*] [–iterate *n*] [–tmout_max *time*] [–tmout_min *time*] [–max *n*] [–min *n*] [–user *username*]... [–user_array *basename*]... [–seed *n*] [–sig *signame*] [–fail_max *n*] [–iterate *n*] [–config *filename*] [–– (*test program args*)] *test_program*

Stress_driver keeps a detailed log, named "stress_driver.log.<pid>" where <pid> is stress_driver's process id. The –**log** option specifies a name for the log file. On Unix-like systems, you can specify /dev/tty as the log file to see the log in your terminal window without cluttering the disk with a log file.

By default, stress_driver will never exit. There are three options and an environment variable that affect stress_driver's immortality. You can use the -life option to define the lifespan of the test run in minutes. Or with this and all the other time options, you can append the letter "s" to the time and it will be interpreted as seconds, which is useful when you're testing the tool. You can also use the stress_time environment variable, which works the same as -life if you don't use the -life option.

Another way to specify the end of a test run is using -**iterate**, which gives a maximum number of iterations of the test that stress_driver will run before exiting. And finally, there's the -**fail_max** option, which gives stress_driver a bit of common sense, so it will exit after encountering the specified number of errors in the test program (either a non-zero exit code or dying from a signal) and any internal errors. You may use all three of these options, and stress_driver will use the first one that applies.

There are also a few options that relate to the lifetime of the test programs. By default, the test programs are not interrupted except at the very end of the stress_driver run. The -tmout_min option specifies that individual invocations of the test program will not be allowed to run longer than the specified time. If you also use -tmout_max, then stress_driver will randomly choose a time between the min and the max timeout each time it starts the test program.

Stress_driver's default action when it decides it needs to stop a test program is to send a SIGINT signal. You can use the **-sig** option to specify a different signal. The signal is specified using its symbolic name, without the "SIG" prefix (like "TERM", "HUP", etc.). The test program may catch the timeout signal if it needs to do any cleanup. It needs to be able to clean up and exit within 30 seconds, or else it will receive a KILL signal.

If the test program starts any child processes, it is responsible for cleaning them up. The test program should not report an error just because it received the timeout signal—this is a normal occurrence. It's okay if the test program simply dies from the timeout signal, though. Stress_driver doesn't log an error in this case.

If you don't specify the **-max** option, stress_driver will start only one invocation of the test program at a time. The **-max** gives the maximum number of test programs to run in parallel. This is really where the tool starts to become useful! If you use -max and not -min, then stress_driver will try to keep the maximum specified number of test programs active all the time. If you use both -max and -min, then stress_driver will first start at the maximum, and then let the load vary randomly between the max and min. It uses an adaptive algorithm to try to keep the average number of active test programs close to the average of max and min. Note that the load may never fall down exactly to the minimum specified.

There is no built-in maximum for the number of test programs that stress_driver can run. In practice, the maximum will be determined by the demands of the test program, and the level of resources that are available on the test system (including memory, the size of the process table, and the processing horsepower). Stress_driver doesn't have any mechanism for distributing the load across more than one test system, though it is conceivable that an intermediary program between stress_driver and the test program could facilitate this with no change to the stress_driver design or to the test program.

The **-seed** option specifies a seed for the pseudo-random number generator. If you don't specify the seed, you'll get different random choices every time you run stress_driver. You can attempt to reproduce the results of a previous run by looking at the seed that is stored in the log file (even if you didn't use the -seed option before) and then feeding that seed into a later run. But later in this paper I'll explain why this isn't very useful.

You can tell stress_driver how to manage user accounts. By default, the test program will run under the same user id as the user who runs stress_driver. If you run stress_driver with administrator privileges, you can use the **-user** option to specify one or more accounts to use instead. If you give more than one account, stress_driver will randomly choose one of them each time it starts the test program. It won't guarantee that the accounts won't be reused for another invocation of the test program at the same time, though.

For more sophisticated account handling, you can use **-user_array**, which specifies the root name of a list of accounts that you've created (like "user1", "user2", "user3", etc.). Stress_driver assumes that the accounts are named using the root name you specify, followed by a number counting up to the maximum number of concurrent test programs allowed. You can use -user_array more than once to give multiple root names (I've done this before to have three different banks of users, with each bank set to a different login shell). Each bank must have enough accounts to handle the maximum load.

Rather than put all of the arguments on the command line, you can create a config file and tell stress_driver where it is using the **-config** option. You can put any command line options and test program options in the config file except for the test program,

described below. Arguments on the command line will override any arguments that are specified in the config file.

After you have specified all of the stress_driver options, you can specify options to send to the test program. First use "--" on the command line to designate the end of the options for stress_driver. Then you may include any arguments that you want stress_driver to pass down to the test program every time the test program starts.

You may randomize the test program arguments either by providing a list of strings, or by specifying an integer range. Here are two randomized test program arguments:

```
[string1 string2 string3] [0-4]
```

Stress_driver will pass two arguments to the test program based on this specification - first, either "string1", "string2", or "string3", and then an integer in the range 0-4 inclusive. Note that in some shells you need to quote the square brackets when using this notation from the command line, though in practice, I generally use a config file when I use randomized parameters.

The final argument on the stress_driver command line is the absolute pathname for the test program. The test program must always be specified as the last argument on the stress_driver command line, not in a config file.

## Stress_factor scaling

There are two different ways to scale a stress_driver run, based on a user-specified stress_factor environment variable, and based on the number of processors in the system.

The stress_factor environment variable is an integer from 1 to 10. (The fact that it's an environment variable and not a command line argument is based on the historical design of the test infrastructure at Convex that ran on top of the stress_driver.) The default stress_factor is 1—this is intended to be a minimal load for the software under test. A stress_factor of 10 is the maximum load that the software can withstand without encountering spurious errors related to resource shortages. For example, you wouldn't want to exhaust the memory on the system unless you're testing memory management. Numbers between 1 and 10 can be gradations in between.

To scale the test based on stress_factor, the test engineer must use a config file, because of the line-oriented syntax of the "factor" lines. Sections of the config file are partitioned using "factor" lines that look like "#Factor n[-m]". The "n" represents a number from 1 to 10. The optional "-m" turns it into a range, like 1-3. All lines after the factor line and before the next factor line will be used if the current stress_factor setting is within the specified range. Thus, the stress_factor scaling involves no magic, just a mechanical way to select options from the config file as they were set up by the test engineer. The engineer is responsible for configuring the test at each stress level, either by scaling the arguments to stress_driver, the arguments to the test program, or both.

Note that you don't have to set up 10 distinct stress levels. In fact, the different stress_factor settings don't even have to have any relationship to each other, but they are set to scale up from 1 to 10 by convention.

As a simple example, consider a case where the test program only takes one argument, "-s", and whatever it does, the argument makes the test more stressful. We will only have two distinct stress levels, so we decide that stress_factor 1-5 will be the low stress level, and 6-10 will be the high stress level.

```
--
# Factor 6-10
-s
```

The "--" tells stress_driver that you're going to list test program options, just like on the command line. Then we have a factor line, telling stress_driver only to use the following lines if the stress_factor is in the range 6-10. We didn't give any test program arguments for stress_factor 1-5, so the test program won't get any arguments when we set stress_factor somewhere from 1 to 5. If the test program is "/usr/bin/testprog", and the config file is named "testprogconfig" in the current directory, we could call stress_driver like this:

```
stress_driver -config testprogconfig /usr/bin/testprog
```

Here's a more complex example from the Convex test suites:

```
# stress_driver configuration for the misc/forker05 test
-fail_max 100
# Factor 1
-max 1
# Factor 2
-max 2
# Factor 3
-max 4
# Factor 4
-max 6
# Factor 5
-max 8
# Factor 6
-max 10
# Factor 7
-max 12
# Factor 8
-max 15
# Factor 9
-max 20
# Factor 10
-max 30
```

The -fail_max argument comes before any of the Factor lines, so it applies to all stress levels. Then we define a different stress level for each of the 10 stress_factor settings, by passing in a different -max argument to the stress_driver at each level. Note that the -max settings do not scale linearly. Stress_factor 10 is 30 times as stressful as stress_factor 1. This format gives us the freedom to scale however we want to.

This 1-10 scaling scheme makes more sense when we look at a suite of tests. The Convex operating system stress test suite was designed to run under the CITE functional test harness. Sometimes we would run a stress test by itself, in which case CITE didn't provide much value. But sometimes we wanted to do a regression test where we would run all of the stress tests for a brief period of time. So we could set the stress_time environment variable to, say, 30 minutes, and we could set stress_factor to, say, 3. Some of the tests recognize both stress_time and stress_factor, some by using stress_driver, and some using other mechanisms. Others may use one or the other. Tests that don't use stress_time are designed to do just one task and then exit. Anyway, with these settings, we'll get uniform coverage across all of the stress tests, at a fairly low stress level, for a fairly short period of time. So we can have global control across all the tests by setting these two environment variables.

## Processor-based scaling

There's another type of scaling that we might want to do on a multi-processor system. A test that is stressful on a single-processor system might not be stressful at all on a system with eight processors. So stress_driver had the ability to scale the test based on the number of processors on the system. Note that this feature is currently not functioning in the version of stress_driver that I'm distributing, because it worked only on systems supported by the "getsysinfo" utility that was part of the Convex test suites. But the infrastructure for doing the scaling is still in the code, and all that is needed is a mechanism to count the number of processors on the system in order to get it working. The code currently assumes that there is only one processor on the system. The mechanism is worth discussing nonetheless.

To use processor scaling, you append the text "xCPU" to an integer argument. Here are two sections from the config file for the shell_stress test that show two types of scaling at work:

```
#Factor 1
-min 1xCPU -max 2xCPU
...
#Factor 10
-min 10xCPU -max 30xCPU
```

On a single processor system, at stress_factor 1, the number of test programs will vary from 1 to 2. If there are 4 processors, at stress_factor 1, the number of test programs running will range from 4 to 8. And at stress_factor 10, with 4 processors, the number of test programs will range from 40 to 120 (10 times 4 to 30 times 4).

You can also use floating point numbers when you use xCPU. The fractional part will be truncated after multiplying, so the result will always be an integer. For example, you might want to use finer control with the scaling like so:

```
-min 1xCPU -max 2.5xCPU
```

So with 1 processor, the range is still 1 to 2, but with 4 processors, the range is 4 to 10.

You may combine xCPU with randomized integer ranges for test program arguments, and you may use fractional numbers here as well.

```
-foo [1-4]xCPU -bar [1-1.5]xCPU
```

The scaling is applied before the randomization, so you get the full range of possibilities. So with 4 processors, the above example is scaled to:

```
-foo [4-16] -bar [4-6]
```

And then the randomization is done within the multiplied ranges.

## Further examples

Here is the first part of the config file for the thread01 test. It illustrates the ways you can get creative with the Factor lines. The arguments to stress_driver have two different stress levels. But the arguments to the test program have ten different levels (the first two are shown here).

```
# Factor 1-5
-min 2 -max 8
# Factor 6-10
-min 4 -max 12
--
# Factor 1
200
# Factor 2
400
...
```

The shell_stress test tries to accurately simulate an interactive user load on an operating system. This is probably the most elaborate use of stress_driver in the Convex tests. The shell_stress tool itself is a fairly complex tool, but it's designed to only simulate one user, so it integrates quite well under stress_driver. For this test, we call stress_driver like so (this is one long line):

```
stress_driver -fail_max 100 -log shell0.log
              -config shell_stress.cf $testbin/shell_stress
```

The stress_driver arguments are split across the command line and the config file for no good reason that I can recall. Here is the full config file:

```
-tmout_min 1
-tmout_max 30
-user test -user cshtst -user shtst -user nettst

#Factor 1
-min 1xCPU -max 2xCPU
#Factor 2
-min 2xCPU -max 4xCPU
#Factor 3
-min 3xCPU -max 6xCPU
#Factor 4
-min 4xCPU -max 8xCPU
#Factor 5
-min 5xCPU -max 10xCPU
#Factor 6
-min 6xCPU -max 12xCPU
#Factor 7
-min 7xCPU -max 15xCPU
#Factor 8
-min 8xCPU -max 20xCPU
#Factor 9
-min 9xCPU -max 25xCPU
#Factor 10
-min 10xCPU -max 30xCPU

#Factor 1-10
--
-seed [1-4294967295]
-shell [/usr/bin/sh /usr/bin/csh /usr/bin/ksh:sh]
-o
```

There are some stress_driver options at the top that apply to all stress levels. Note that I took advantage of the free-form format of the file to try to make it more readable. I specify four different user accounts to choose from. These were standard accounts that were always set up on systems that were configured to run any of the operating system tests.

The -min and -max arguments to stress_driver are scaled based on the stress_factor and the number of processors, as described earlier. Then at the bottom of the file, I specify the test program arguments that don't scale on stress_factor. I likely forgot the "#Factor 1-10" line when I first wrote the config file, and was surprised to find that my test program only got its options at stress_factor 10, since the "#Factor 10" line is still in effect until the next Factor line.

For the test program arguments, I set the pseudo-random seed for shell_stress. This is based on a random choice across a wide range, and that random decision in turn is based on stress_driver's seed. This was an attempt to make the test run reproducible, so that all random decisions at all levels are tied to stress_driver's seed. Note that I

didn't use stress_driver's -seed option here - we just let stress_driver randomize the seed. It's easy to confuse the two different drivers here. For our big reliability test, which had a somewhat different setup, we did hard-code a seed for stress_driver, using a large prime number.

In practice, I found that the results for two different shell_stress runs with stress_driver using the same seed weren't necessarily the same. Keep in mind that the types of bugs that shell_stress found often depended on exact timings that occurred by random chance more than intentional test design. Even if we ignore that factor, just comparing the logs from two stress_driver runs with the same seed show that stress_driver wasn't making the same random decisions in both cases.

Why did the seed not do what I wanted it to? I haven't studied the reason in depth, but here's a theory. Complex computer systems are not completely deterministic. When we have hundreds of processes running, there is no guarantee that they will exit in the same order each time. Perhaps the disk is fragmented in a different way and its response time is different, or perhaps you ran a command on the system that was the equivalent of a butterfly flapping its wings and changing the weather on the other side of the globe. In any case, as soon as a stress_driver action is done in a different order than the previous run, then the next number in the pseudo-random sequence may be applied for a different purpose than for the last run. Then the place where that number was used last time gets a later number in the sequence instead. That's all it takes for the test run to skew wildly. Further study would be needed to figure out how to prevent this, and whether identical behavior from stress_driver is likely to have much effect on reproducing failures in the first place.

The "-shell" argument is a use of the string type of randomization, telling shell_stress which shell to use. You may have noticed that the names of some of the user accounts also suggest a type of shell—these are the login shells for the accounts. Neither stress_driver nor shell_stress (in this particular test) does a full login, so the shell is chosen independently of what the login shell for the account is. The “:sh” notation tells shell_stress to use Bourne shell-style syntax when setting the shell prompt and checking that status of the commands.

## Porting stress_driver

Now we can fast forward to 2002. Eric Schnoebelen, another ex-Convex employee, was doing contracting work for the Hewlett-Packard division that had acquired Convex. HP was no longer actively using the test suites that it had acquired with Convex. Eric convinced HP to release the tests and their associated tools under an open source license. Eric volunteered his time to audit the tests to remove the functional tests that Convex has licensed from Perennial, and the tests are now available, along with the CITE test harness that many people had requested a copy of during it heyday.

I decided to pull out one particular part of this valuable but obscure resource, and help others take advantage of it. So I ported stress_driver to the Cygwin environment on Windows (a library that facilitates porting Unix utilities to Windows, plus the Unix-

style utilities that use it), and Linux. I did most of the work on Windows because I had a Windows system handy. When I tried the tool on Linux for the first time, it required no changes in order to run properly. The tests, however, required some significant porting work because of the way I had designed them.

I adopted one tenet of extreme programming, and did a sort of test-first development, or in this case, test-first maintenance. I published a brief write-up about this effort, in the article "Test-First Maintenance", which is included at the end of this paper.

One of the first things to go from the stress_driver code was the references to several "h2ph" files. These files are produced by a script that tries to convert C header files into Perl. I referenced some of these files for the advanced signal handling that is required to support the event-driven aspects of stress_driver. These perl headers were very fragile, and I had to work around a few bugs in them. Another big hack in the code was a use of the "syscall" function to invoke a system call directly from perl, also for the purpose of advanced signal handling.

I wanted to rip out my home-grown event-handling code and use something like the Event module instead. The Event module is part of the Comprehensive Perl Archive Network (CPAN), though it doesn't install with Perl by default. I was apprehensive about removing my event handling code. I had put a lot of work into making it robust, and it was a core part of the code, though there was still an occasional mysterious failure. I decided to put off the port to the Event module for a while.

Since I wanted to make the script portable, I decided to port the signal handling code to use POSIX signals. The POSIX module was not available when I first wrote stress_driver using Perl 4. Using the POSIX module would not only make the code more portable, but it would also get rid of the dependence on the most egregious hacks in the code—the h2ph files and the use of the syscall function. I had the port partially done, and at the same time I was writing automated tests to verify the stress_driver code. I found that one of my tests was failing intermittently. It looked like I had a race condition in my code. At this point, I decided it was time to bite the bullet and make the big changeover to the Event module rather than trying to fix the existing code.

The changeover wasn't as traumatic as I had feared. I ended up removing about 90 lines of code that were replaced by functionality in the Event module. I still need to do more testing to verify that stress_driver is working as well it used to, though.

There are some other Perl 4'isms that I've been working on removing. To parse the command line arguments, I used the NGetOpts function from the newgetopt.pl library, which was the latest and greatest method for argument handling in Perl 4. I used NGetOpts to get stress_driver's command line arguments, and I also crafted a hack to use NGetOpts to parse the config file. I have ported the code to use the GetOptions function in the Getopt::Long module instead. So some nasty perl 4 hacks with "package" are replaced with some "write-only" code that deals with the hash that now

stores the arguments. A bit of a hack is still required to convince GetOptions to process the config file, but it's not nearly as crufty.

Another Perl 4 relic was the fact that I localized my variables using local(), which uses dynamic scoping. For Perl 5 programs, programmers are strongly encouraged to use my() instead of local(). The my() syntax specifies lexical scoping, which is a safer and much more familiar mechanism (even if you don't know what lexical scoping is). But for the config file hack mentioned above, I found that I still had to use "local(@ARGV)" because GetOptions references @ARGV as a global variable. When I naively tried "my(@ARGV)", the value wasn't available to GetOptions because of the lexical scope.

# Limitations

While stress_driver was written to be general-purpose, it's not likely to be appropriate for everyone. It was designed for operating system testing, and it runs directly on the system under test. So there are no special features for starting the application under test.

Stress_driver is Unix-centric, and it doesn't have a graphical user interface. Though it runs under Windows with a lot of help from the Cygwin environment. someone who isn't familiar with Unix or Cygwin may have trouble using the tool.

You can only specify one test program to stress_driver. If you wanted to use more than one test program, you could write a wrapper program that called your test programs using whatever criteria you wanted. In fact, you could consider the shell_stress test to be an example of this. Shell_stress runs many different programs from its user profile database. The downside is that all these layers of control steal performance from the system (and shell_stress itself is two layers - a perl script to parse the database and an expect script to execute the commands). While running the shell_stress test, I found that the system spent a sizable fraction of its resources running all the driver scripts. This takes resources away from the tests themselves. If you write a test driver specifically for a particular type of test, you have more opportunity to optimize the driver. This is the tradeoff we make for a general solution.

The performance issue might could be mitigated if stress_driver could execute tests in a distributed fashion, so that one machine executes the driver and other machines execute the tests. It is possible, of course, to add a layer underneath stress_driver that distributes the tests, which would bring with it all the caveats of the previous paragraph. For one incarnation of shell_stress, I did add such a layer. I only used it to test networking in a loopback, i.e., doing telnet, rlogin, and ftp back to the same machine, but it did serve as a proof of concept for doing distributed testing.

Another factor to consider is that each invocation of the test program requires starting a new process. This wouldn't be ideal for situations where the test cases are very lightweight. For example, if the test runs in a tenth of a second, then the time required to start a new process, clean up after it, and log each step along the way would dwarf the time spent actually running the test code.

## Ideas for enhancement

There are always more features to implement that resources available to implement them. Here are a few ideas for stress_driver that may also give you ideas for improving other tools that you work with. Whether any of these get implemented will depend on the interest from the user community and on how many people volunteer to help with the development.

- *Increase test coverage.* The current test suite is very sparse and there are likely many more bugs to root out.
- *Port and test on other systems.* It's likely to port easily to any system that supports Perl and the Event module (probably only Unix-like environments).
- *Fix the cpu scaling feature.* For each supported operating system, it just needs to have a mechanism to count the number of processors on the system. Also, perhaps add a command-line option to specify the number of processors, which could be used before the automatic processor count is ready, and could also be used to spoof the number of processors for purposes of experimentation.
- *Add a graphical interface using Tk.* This would make stress_driver easier to set up and monitor.
- *Validate the math used in the adaptive scheduler.* I suspect that it doesn't quite work the way it's supposed to, in managing the average number of active test processes.
- *Provide hooks that would allow dynamically modifying the stress_driver and test program arguments.* Users could use their own adaptive algorithms.
- *Improve process management.* Several possibilities here, such as: using process groups to make cleanup more robust, modify priorities so stress_driver gets more cpu cycles when under heavy load, and test stress_driver's operation when the process table or memory is full.
- *Implement a slow start feature.* Rather than always blasting the system with the maximum number of tests all at once, it might be useful to be able to start up more slowly in order to mimic more of a real-world scenario.
- *Allow math expressions in arguments.* The current scaling mechanisms are fairly flexible, but we could get even more flexibility by allowing arbitrary expressions and further generalizing the scaling.

## Call for participation

Consider yourself invited! Perhaps you want to be a user of the tool, or you just want to borrow the code for other purposes. You could hone your test automation skills by contributing to the stress_driver test suite, or you could exercise your perl programming skills by working on the stress_driver code itself. Both the original code from Hewlett-Packard and my enhanced version of the stress_driver is released under a modified Apache Project license.

To obtain the enhanced version of the stress_driver and its test suite, go to http://www.tejasconsulting.com/stress_driver/.

To run stress_driver on Windows, you will need to install Cygwin and the Cygwin build of Perl. See http://www.cygwin.com/. On Linux and Unix systems, you will need a recent version of Perl. I used Perl 5.6.1 on both Windows and Linux.

You will need the optional Perl modules: Event, Test::More, and probably an upgrade of Test::Harness. The easiest way to install these if you have a live Internet connection is to run "perl –MCPAN –e shell" and type "install Event" and "install Test::Harness". Note that on Windows 2000, one of the Test::Harness self-tests will fail, so you have to do a forced install.

For the original stress_driver as of the time of its 2002 release from HP (the script was actually last modified in 1996), including the suite of stress tests that used it, see ftp://ftp.cirr.com/pub/cite/test-suites-19961217.tar.gz. The script is located in bin/stress_driver, and the test suites are under os/stress. This version of the script is also included in my stress_driver distribution, named "stress_driver_orig".

The test suites are designed to run using the CITE test harness, which can be found at ftp://ftp.cirr.com/pub/cite/cite-4.4.tar.gz. Stress_driver itself is not dependent on CITE.

Note that the originally released testware in test-suites-19961217.tar.gz and cite-4.4.tar.gz run only on a limited set of now outmoded platforms. It's difficult even to determine which platforms they did run on when development ceased. So don't expect to be able to make use of what you find there without porting it to your platform.

# Acknowledgments

Big thanks go to Eric Schnoebelen for taking the initiative and applying the elbow grease to get the stress_driver code and megabytes of other interesting test assets released to the public, and to Hewlett-Packard for agreeing to make it available. Thanks also to the System Software Test Team at Convex Computer Corporation for their role in the development of stress_driver, the infrastructure it worked within, and all the suites that used it.

I'd like to acknowledge Bob Clancy for reviewing early drafts of this paper and for being to first to volunteer to participate in the further development and testing of stress_driver.

# References

"Test-First Maintenance: A diary", Danny R. Faught, Dallas/Fort Worth Unix Users Group Newsletter, June 2002. (included below)

Event-Driven Scripting, Danny R. Faught, presented at the July 12, 2001 meeting of the Dallas/Fort Worth Unix Users Group. Slides at: http://tejasconsulting.com/papers/event-driven/Event-Driven.htm.

“Perl Scripting: A Test Automation Task Master”, Danny R. Faught, Software Test Automation Conference tutorial, Fall 2002. Includes a walkthrough of some of the stress_driver code.

# *Appendix*

## Test-First Maintenance: A Diary

Published in the Dallas/Fort Worth Unix Users Group Newsletter, June 2002

Hurrah! A stress test tool that I wrote while employed at Convex Computer Corporation has been released with an open source license. It's called "stress_driver" and it's sitting hidden in a 20 meg tar file where no one will likely find it, and anyone who does find it won't know what to do with it. It runs on an operating system version that few people use. It was written using Perl 4, and though it's been ported to Perl 5, it still uses a Perl 4 style, including requiring some header files that are conjured via black magic. But I found that it was a very useful tool, and I bet that it could easily be ported to other operating systems.

I've been talking to Extreme Programming (XP) and other agile development advocates about test-first development. So why not test first maintenance? The idea with test-first development is that when you develop a new feature, you first write a test, you run the test to verify that it fails, you develop the feature, then run the test and see if it passes.

Here I have an 816-line perl script that doesn't run on any system I have access to. There are no tests. I'm going to dive into the deep end and try test-first maintenance for legacy code, while porting stress_driver to the Cygwin environment on Windows NT 4.0. I'm keeping a diary along the way. Here are some highlights and extra commentary.

Oh, by the way, I'm not familiar with Perl's test harness modules, though I know that several exist. Having run the test suite for Perl itself and some of its modules, I choose the same basic "Test" module that they use, and I decide to use Test::Harness in a script that will kick off all of the tests.

**2002-05-08**
**2:08pm**
The simplest test I can write is one that uses no command line arguments. It turns out that this is a negative test - the expected result is an error message, because at least one argument is required. I don't think agile developers write a lot of negative tests. Oh well. I write the "badopt" test. It passes, but I didn't verify the text of the error message. It turns out that the stress_driver is croaking because I haven't starting porting it to Cygwin yet. So I add another check based on the error message, and that fails.

Seems like I have a lot of work to do to get this first test to pass. Hmmm, the XP tenets say I should keep things simple. So I simply comment out the parts that don't work and are preventing the program from getting as far as the code that checks the command line arguments. I have my first passing test!

I decide to add some subtests to "badopt." As a tester, I find negative tests more fun than positive tests. :-) One new test passes a test program path to stress_driver that doesn't exist. (Talking about testing a test tool can be confusing - when using stress_driver, you give it the path of a test program that it will run) Cool, that passes.

**2:24pm**
I want to add another test that specifies a program that isn't executable. Now is when I start wishing for a more complete test environment. I need to create a file and make sure the execute bits are off. Normally, I expect the test harness to give me a working directory where I can create any files that are needed. I hack my test harness script so it creates a working directory and put the path in an environment variable. That subtest passes. In retrospect, I wonder why I didn't just create a non-executable file ahead of time in the test suite directory. Maybe because it it's too easy for file permissions to be botched when installing a test suite.

**2:45pm**
Okay, I'll force myself to write some positive tests. I create the second test, named "simple." I'll tell the stress_driver run "sleep 100000" and then interrupt it shortly after it starts. There is a -life option that tells stress_driver how long to run. Unfortunately, the lowest it can go is one minute, which is unacceptable for a test case that should be able to do its job in a few seconds. I modify stress_driver so that the -life option can understand seconds as well as minutes.

Testers often have to ask developers to add testability features to their programs. It's such an easier sell when I'm both the tester and the developer. I recall when I originally developed the code, I modified it so that minutes were interpreted as seconds while I was testing, but since I didn't write any reusable tests, I didn't bother to support both.

**2002-05-11**
**9:33am**
A big change that I've been planning to make is to rip out my home-grown event loop and use the Event module instead. I now have 12 subtests in three files, 15 seconds runtime. All usually pass, but one intermittently fails in the event code. I decide it's time to do the big changeover rather than trying to fix the old code.

**2002-05-13**
**2:41pm**
All tests are now passing after the event code changeover (and the code is about 90 lines leaner now). But I'm suspicious - that was too easy. I examine the logs created from running the "simple" test, and I see that stress_driver never actually started any test programs. My tests need to do a lot more verification. I realize that I'm using a unit testing framework to do high-level functional testing. Verification would be much easier and more thorough if I were doing true unit testing and had more access to the program state.

**2:49pm**
Oops! I learn that when commenting out some of my event code libraries, I also commented the code that initially starts the child processes, which is still needed in the new event design. Fixed. I'm glad I tend to comment out code and test the program before actually deleting the code.

**10:05pm**
Fixed several other problems, and the post-Event module code now passes all 12 tests.

**2002-05-17**
**5:19pm**
I'm getting tired of setting -life to one second for my positive tests. It's not elegant, and it's still not as optimized as it could be. So I give stress_driver a new -iterate option that specifies the maximum number of times to iterate the test program. For many of my stress_driver tests, I'll specify just one iteration, and many will complete in less than a second. I wonder why I never thought to create that feature before. Chalk up another one for testability.

Well, that's where I'll leave you for now. Along the way, I found bugs in my original stress_driver design (including a minor Y2K bug) as well as the new code I added. I found bugs in the Event module and perl itself, including a reproducible crash in the perl interpreter. I found myself wishing for a more full-featured test environment, so I plan to investigate the other Test modules that are available.

If you're a Perl hacker who's interested in participating in the test-first maintenance project and in using an alpha version of a general-purpose stress test tool, let me know. There's plenty more testing to be done.