Journal of Zhejiang University SCIENCE A
ISSN 1673-565X (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

A recoverable stress testing algorithm for compression and encryption cards*

Bao-jun ZHANG†, Xue-zeng PAN, Jie-bing WANG, Ling-di PING

(*Network and Security Lab, School of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China*)

†E-mail: zbjhover@zju.edu.cn

Received Dec. 17, 2007; revision accepted Apr. 8, 2008

Abstract: This study proposes a recoverable stress testing algorithm (RSTA) for such special devices as compression/decompression card and encryption/decryption card. It uses a chaos function to generate a random sequence, and then, according to the random sequence, generates an effective command sequence. The dispatch of command obeys a special schedule strategy we designed for such devices, i.e., the commands are sent according to the command sequence, and the complete commands are put in a buffer for further result check. RSTA is used to test the HIFN compression acceleration card SAICHI-1000. Test results show that RSTA can make the card work continuously and adequately.

Key words: Stress testing, Random sequence, Chaos function, Synchronization, Concurrency
doi:10.1631/jzus.A0720130 **Document code:** A **CLC number:** TP312

INTRODUCTION

Stress testing, used to determine the stability of a given system or entity, is an effective method for improving product reliability and is very important to product development. Some defects of products, which are not easy to find in manufactory, can be detected through a strict stress testing. The quality of a product can be guaranteed by this testing, whose benefits have been introduced in detail by Chan (1995).

Since the reliability of a product determines its market, the stress testing is increasingly being adopted (Hobbs, 1987; Shinner, 1988; Pachucki, 1994). However, these tests are mainly empirical and are designed for some specified products, so a direct extrapolation of the application from one product to another may mislead the interpretations. Hobbes (1992) introduced different stress regimes to guide the stress testing process. Barber and Gehner (1992) proposed an economic analysis of stress testing. Elbert *et al.*(1994) analyzed the relationship between stress testing and product reliability. McLinn (1998) proposed some methods to improve the analysis of step-stress testing, which has many worthwhile applications in the analysis of the projected life for components and occasional systems.

All above works help to understand the concept and guide the processing of stress testing. In recent years, many stress testing methods have been designed for new applications. Cui *et al.*(2002) designed a stress testing method to test the OSPF (open shortest path first) protocol on large-scale routing simulation. Cui *et al.*(2003) proposed a novel stress testing method to test the running characteristics of routing protocol implementation (RPI) in a real large-scale network. Liu *et al.*(2006) introduced a system model of CDMA (code division multiple access) stress testing service platform based on state machine technology. This system provides an efficient service-oriented solution to protocol and stress testing of a CDMA system.

* Project supported by the Hi-Tech Research and Development Program (863) of China (No. 2006AA01Z431) and the Giant Project of Zhejiang Province, China (No. 2006C11105)

Stress testing can be used for both software and hardware. Roy *et al.*(1995) used stress testing in combinational VLSI circuits. The technique, based on reordering of test vectors, generated a desired circuit activity or electrical stress across the VLSI chip while achieving a high coverage for stuck-at defects. By now, accelerated stress testing (AST) has been used in electronic, electromechanical and mechanical systems to achieve robustness with high reliability primarily for hardware (Chan, 2004). AST is also used to detect probabilistic software failures in (Gullo and Davis, 2004).

When stress testing is used for hardware, the throughput of stress testing should be large enough to follow the hardware speed. Or else, the characteristics of hardware can be hardly obtained. In addition, the test system for a longer period of time helps to find faults of a system, since under normal conditions certain types of bugs, such as memory leaks, can be fairly benign and difficult to detect over the short periods of time in which testing is performed. Here we propose a recoverable stress testing algorithm—RSTA, which is designed for the products of HIFN, such as compression/decompression card, encryption/decryption card, and can be used for other similar devices. RSTA is of high throughput and can work continuously. It is widely used in our product test.

The rest of this paper is organized as follows. Section 2 introduces the RSTA. Section 3 analyzes the algorithm in detail and gives an evaluation to the algorithm. Section 4 concludes the paper.

RSTA

Design of RSTA

RSTA aims at devices such as compression/decompression card, encryption/decryption card, etc. Before introducing the RSTA, we should know the features of these devices first. Take the HIFN compression acceleration card SAICHI-1000 as an example. In order to make it work fully, we should do continuous compression and decompression with different block sizes and data. Every now and then, we need to add some error conditions to conduct a negative test. There are two necessary conditions when we do stress testing to the compression card:

Condition 1 (Command sequence) For the same data, the decompression should start after the corresponding compression has been completed.

Condition 2 (Command dispatch) The test must run enough time to be an adequate test. Since the resource of a computer is limited, especially the memory is exhausted quickly, we have to reuse the memory for data compression/decompression to make an adequate test. When the memory needs to be reused, we should wait until the decompression and the corresponding data check are completed.

Based on these two conditions, the algorithm can be divided into four parts: random sequence generation, command sequence generation, command dispatch, and result validation, which are described one by one in detail below.

1. Random sequence generation

First, we generate a random sequence, which is a binary sequence containing only '0' and '1'. We do not need to generate a strict random sequence as required in the cryptology. Since the chaos function can generate a large quantity of random data at one time and the logistic map is easy to implement, we use the logistic map to generate the random sequence:

$$x_i=px_{i-1}(1-x_{i-1}),\ x_i\in(0, 1),\ i\in\mathbb{N}^*. \tag{1}$$

After about 100~200 times of self-iterations, the logistic map will enter the status of chaos, and the resulting data sequence is a random sequence (Bose and Banerjee, 1999).

Fig.1 shows the chaos phenomenon generated by the logistic map. According to Fig.1a, when p=4, the logistic map will achieve the best chaos effect. Fig.1b shows the random sequence when the map enters the chaos status with p=4 and the initial value x_0=0.391.

In order to recover the stress testing, the random sequence should be controlled. A seed s should be provided as an initial parameter of the chaos function:

$$\text{DWORD } s;$$
$$x_0=s/2^{32},\ s\in[1, 2^{32}-1]. \tag{2}$$

Eq.(2) makes sure $x_0\in(0, 1)$.

Then we convert these random data into 8-bit binary integers according to Eq.(3) to obtain the random binary sequence:

$$\text{BYTE } y;$$
$$y=(\text{BYTE})(x*1000)\%256. \tag{3}$$

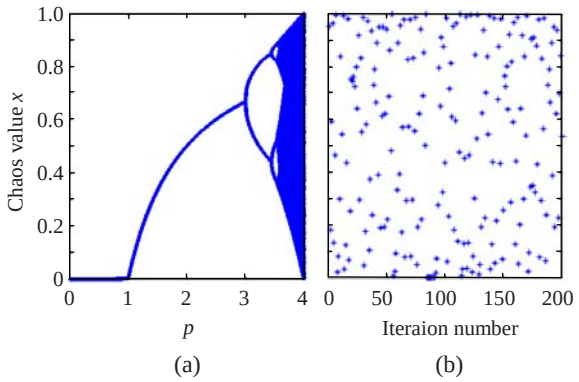

Fig.1 (a) Chaos phenomenon generated by the logistic map. The best chaos effect is achieved when p=4. (b) The random sequence when the map enters the chaos status with p=4, x_0=0.391

2. Command sequence generation

In this step, we will generate the command sequence according to the binary random sequence. It is the key step of RSTA. Fig.2 shows the details to generate the command sequence, where S*i*, C*i* and D*i* (*i*=1, 2, ..., 16) stand for session *i*, compression *i* and decompression *i*, respectively. D*i* is the decompression corresponding to C*i*.

A session is a pair of commands—compression and decompression. For each session, we allocate three memory buffers and fill the original data into the source buffer of compression. The three buffers are:

(1) Source buffer for compression—used to store the original data.

(2) Destination buffer for compression & source buffer for decompression—used to store the compressed data. Since the data in the source buffer for decompression are the same as those in the destination buffer for compression, these two buffers are in fact the same.

(3) Destination buffer for decompression—used to store the decompressed data. The data should be identical to the original data in the source buffer for compression. In stress testing, the decompressed data need to be checked.

For example, there are 16 sessions and totally 32 commands: {(C1, D1), (C2, D2), …, (C16, D16)}. We need to generate two 8-bit binary integers. For example, we obtain two numbers (107, 197) from the first step. The binary format of these two integers is shown in Fig.2a.

First we rearrange the sequence of the sessions. Refer to Fig.2b, we put the sessions corresponding to '0' together in the front of the session sequence, and those corresponding to '1' in the back of the session sequence.

Next, we generate the command sequence. There are two issues to be addressed: the sequence of commands and the concurrency of commands.

(1) How to ascertain the sequence of commands?

The sequence of commands should obey Condition 1. That is to say, the decompression command should follow the corresponding compression command, while the compression command reusing the memory buffer should follow the previous decompression command using the same buffer. According to the session sequence in Fig.2b, to meet Condition 1, for each session we send compression commands no matter the binary number is '0' or '1' in the binary random sequence. Then we set a condition to dispatch decompression. For example, in Fig.2c we dispatch decompression command when the current bit is '1'. We dispatch both compression and decompression commands according to the session sequence. This makes sure that the decompression command follows the corresponding compression command.

When we send decompression commands, the concurrency of the commands should be taken into account. It determines whether our algorithm can make hardware work fully.

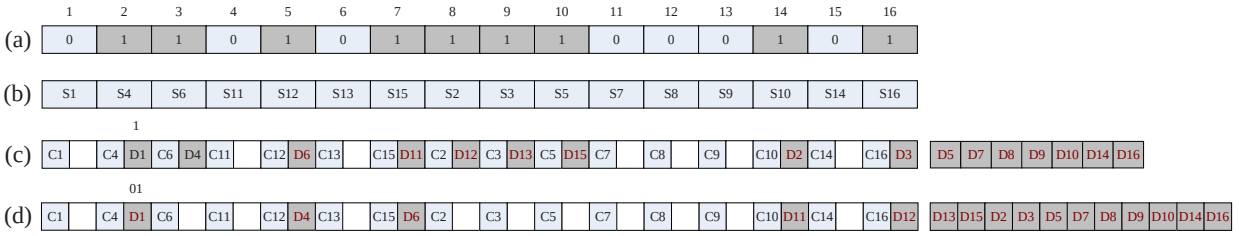

Fig.2 Command sequence generation. S=session, C=compression, D=decompression. (a) Random sequence; (b) Session sequence; (c) Command sequence with lower concurrency; (d) Command sequence with higher concurrency

(2) How to enhance the concurrency of commands?

Fig.2c shows the schedule strategy with a lower concurrency. When the bit in the current binary random sequence is '1', the corresponding decompression command should be dispatched. Since the process of decompression should wait until the corresponding compression is completed, the schedule strategy shown in Fig.2c is not good and the concurrency of commands is lower. Fig.2d shows a command schedule strategy with a higher concurrency, which makes dispatch decompression more difficult. That is, only when the previous bit is '0' and the current bit is '1', will the decompression command be dispatched. It is obvious that the number of continuous compression commands is bigger than that in Fig.2c. In actual applications, we can adjust the difficulty of decompression command dispatch according to the processing ability of the hardware.

We prepare the sessions, such as memory allocation and data preparation, before the test. Then we generate the command sequence. The sessions are used repeatedly so that the memory can be reused. We do multi-round test, for each round, with a different command sequence. Thus we can keep the test continuous with a limited memory.

3. Command dispatch

For example, we do two-round test with four sessions {(C1, D1), (C2, D2), (C3, D3), (C4, D4)}. We obtain the command sequence {C2C1D2C3C4 D1D3D4, C1D1C4C2C3D4D2D3}. There are two details for command dispatch. First, when D2 is dispatched in the first round, we should wait until C2 is completed. Second, when C1 is dispatched in the second round, because the memory of S1 is reused, we should wait until the data validation of the previous round on this session is completed.

4. Result validation

In another thread used for the result validation, we compare the source buffer of the compression with the destination buffer of the decompression in the same session. Then log the result.

Implementation of RSTA

RSTA has been used for SAICHI-1000 under Windows 2003. SAICHI-1000 is a lossless data compression card produced by HIFN. It uses the PCIe bus and has four 9630 compression chips. Each chip can provide 300 MB/s throughput. Fig.3 shows the framework of stress testing. Fig.4 shows the flowcharts of each function module in RSTA. To implement RSTA, we have developed the following procedures:

(1) Asynchronous schedule mechanism. Asynchronous schedule mechanism can enhance the concurrency of commands. The dispatch thread does not have to wait till the command is completed. We always use asynchronous mechanism to make the hardware work fully.

(2) Reuse of the memory buffer. There are two ways to save memory. One is reusing the memory allocated in advance. The other is releasing and reallocating the memory. We adopt the first way to lower the complexity of the algorithm and save the time used for memory release and reallocation.

(3) Session preparation before the test. Session preparation is composed of memory allocation and data generation. Sessions are prepared before the test to make it coherent and need no more data processing during the test.

(4) An individual thread for result check. The result of each session should be checked. An individual thread is created to check the result. We separate the result check from the dispatch thread, which advances the command dispatch. For communication between the checking thread and the dispatch thread, we use a queue to store the complete sessions. A complete session means that the pair of commands in that session has been completed. The dispatch thread puts the complete session in the tail of the queue, and the checking thread gets the session from the header

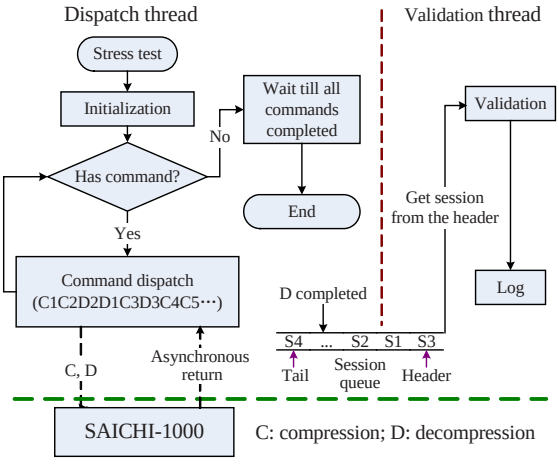

Fig.3 Stress testing framework of SAICHI-1000

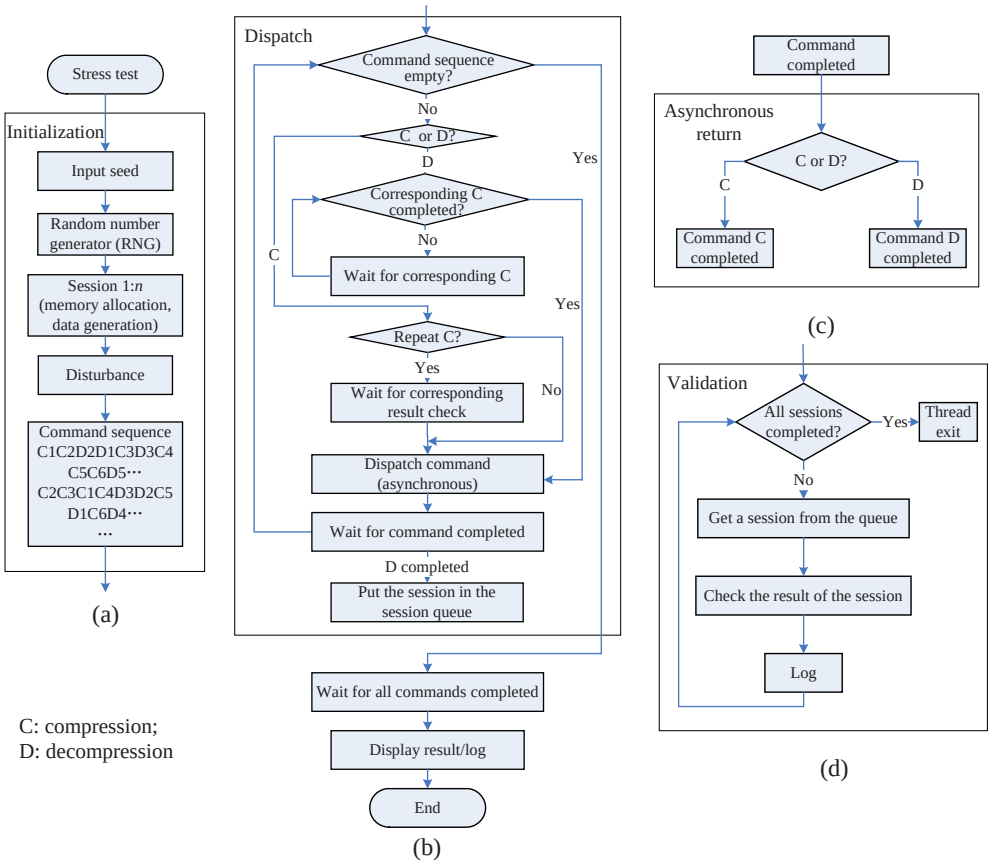

Fig.4 Stress testing module. (a) Initialization; (b) Command dispatch; (c) Command return; (d) Validation

of the queue. A mutex lock is used to protect the queue. Two events are used to synchronize these two threads. One is triggered in the dispatch thread to notify the checking thread that the queue is not empty, so the checking thread can get the complete session from the queue. The other is triggered in the checking thread to notify the dispatch thread that the queue is not full, so the dispatch thread can put the complete session in the queue.

(5) Increase the concurrency of commands. Two methods are used to increase the concurrency of commands. One is the asynchronous schedule mechanism mentioned above. The other is adjusting the difficulty of decompression command dispatch to generate a more adaptive command sequence with a higher concurrency.

ALGORITHM ANALYSIS AND EVALUATION

In software test, stress testing refers to the tests that determine the robustness of software by testing beyond the limits of a normal operation. Stress tests commonly put a greater emphasis on robustness, availability and error handling under a heavy load than on what would be considered correct behavior under normal circumstances.

The software being tested is ‘mission critical’, that is, a failure of the software would have disastrous consequences. Under normal conditions, certain types of faults, such as memory leaks, are difficult to detect over the short periods of time. Stress testing should make the test durable and full for finding these latent faults. Since the resource dedicated to the testing is usually not sufficient, how to make a test adequate with limited resources is critical to the design of stress testing.

Take the compression card as an example. Since the compression card is used to compress data, and the operation of hardware is faster than that of software, to make hardware work fully, we should enhance the concurrency of command dispatch. So the following problems should be considered.

(1) What is the “mission critical”?

For the compression card, it is obvious that the compression/decompression command is the “mission

critical".

(2) How to make the test durable with limited resources?

The main computer resources are the memory and CPU. In general, people allocate and then release the limited memory when it is not sufficient for other applications. This method will complicate the algorithm and occupy more CPU clock cycles due to continuous memory allocation and release. The other way is to pre-allocate enough memory and to use them repeatedly until the program ends. We choose the second way to allocate memory for all sessions in advance, to prepare data for these sessions, and to repeatedly dispatch these sessions through different command sequences. We do not have to free the memory when doing stress testing, which avoids the disadvantages in the first way.

For saving CPU clock, we cut such trivial operations as avoiding allocation and de-allocation of memory during stress testing and only operate data compression and decompression; to enhance the CPU utilization rate, we create an individual thread to do result validation and log, and enhance the concurrency of commands.

(3) How to enhance the concurrency of executions?

As mentioned in Section 2, the asynchronous schedule mechanism is used, and based on Conditions 1 and 2 we generate an appropriate command sequence by controlling the dispatch of decompression commands according to the number of commands that can be executed at the same time.

(4) How to make the test recoverable?

RSTA uses a seed as the initial value of the logistic map to generate the random number sequence. Then it generates the command sequence according to the random sequence. The seed and the number of sessions can be used to recover the command sequence. The seed is also used to recover the data using the C language function *srand*() when compressing the random data.

RSTA has been implemented and tested on Dell 840 with SAICHI-1000. Figs.5a and 5b describe the stress testing results for 6 seeds and 97 652 438 seeds, respectively.

In Fig.5, 'loop' refers to the run times of the test. In order to make stress testing durable with limited resources, we can do stress testing with the same

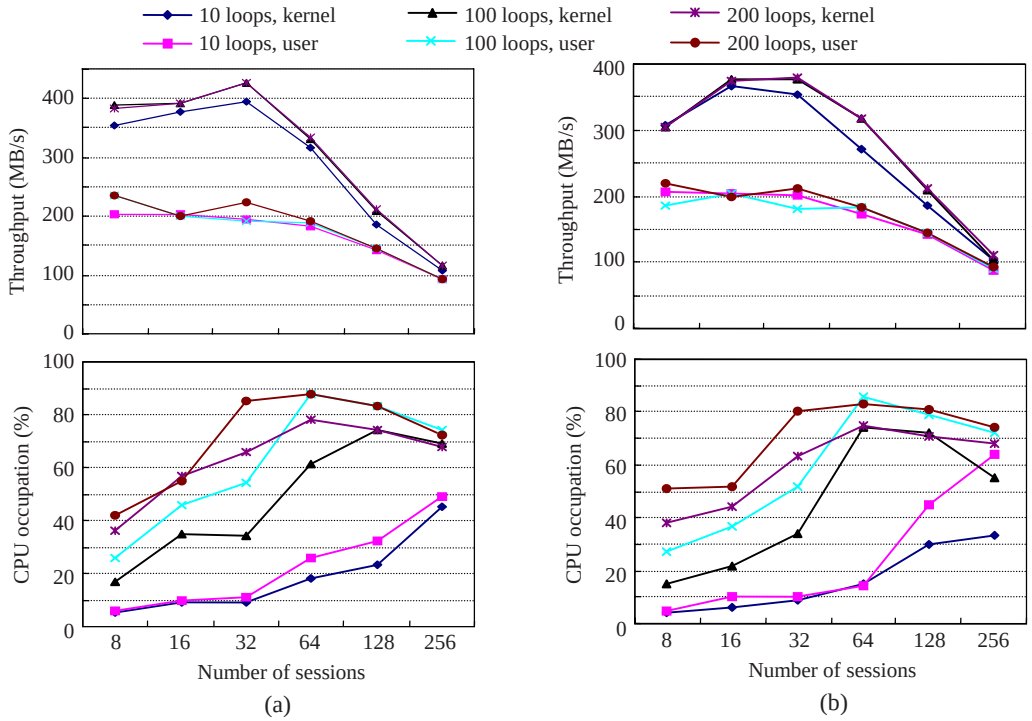

Fig.5 Throughput and CPU occupation for different numbers of seeds when the number of sessions changes from 8 to 256. (a) 6 seeds; (b) 97 652 438 seeds

number of sessions repeatedly in different command sequences for each round. The memory type 'kernel' refers to memory allocated from kernel reserved memory pool; 'user' refers to memory allocated from user space.

We calculate the throughput with the total time of stress testing and the total size of input data of compression and decompression.

We can see from Fig.5 that

(1) The highest throughput of stress testing reaches 427 MB/s.

(2) The 'kernel reserved memory' has a higher throughput than the 'user space memory'.

This is because the driver of SAICHI-1000 uses DMA to transmit data. When the data are in user space memory, we need to copy the data to the kernel consecutive memory. So the user space memory has a lower throughput. The memory copy can be avoided in DMA transmission when the hardware supports the mode of 'scattering-gather'. The user space memory will achieve the same performance as the kernel reserved memory.

(3) As the number of sessions increases, the throughput of stress testing increases first and then falls. The throughput reaches the highest value when the number of sessions is 32.

More sessions will occupy more resources and cost more time in management. So the throughput is not possible to increase unlimitedly as the session number increases.

(4) As the loop number increases, the throughput does not change obviously, but the CPU occupation significantly increases.

As the loop number increases, the stress testing will last longer and the test is more sufficient, which leads to more CPU occupation.

(5) User space memory has a higher CPU occupation than kernel reserved memory.

Because user space memory needs to copy data when the driver does DMA transmission, more CPU time is wasted.

(6) The number of seeds has minute effect on the test results in both throughput and CPU occupation.

Different numbers of seeds generate different command sequences, which lead to different results. But the seed is not the determinative reason to affect the throughput, so the change is minute.

In short, RSTA has a high throughput up to Gbps level, and the results vary with the change of test parameters. The variety of RSTA makes the system test adequate and helps to find the faults, and thus guarantees the product quality.

CONCLUSION

This paper introduces in detail the recoverable stress testing algorithm, RSTA, aiming at devices such as compression card and encryption card. Test results demonstrate that RSTA is recoverable and can make the test durable and full with limited resources. It has been used to test the compression card—SAICHI-1000 with good performance.

References

Barber, J.S., Gehner, K.R., 1992. Age-and-test procedures for true reliability improvement, feasibility and economic justification. *IIE Trans.*, **24**(5):81-87. [doi:10.1080/07408179208964247]

Bose, R., Banerjee, A., 1999. Implementing Symmetric Cryptography Using Chaos Functions. Proc. 7th Int. Conf. Advanced Computing and Communications, p.318-321.

Chan, H.A., 1995. The benefits of stress testing. *IEEE Trans. on Comp., Pack., Manuf. Technol., Part A*, **18(**1):23-29. [doi:10.1109/95.370730]

Chan, H.A., 2004. Accelerated Stress Testing for both Hardware and Software. Proc. Annual Reliability and Maintainability Symp., p.346-351. [doi:10.1109/RAMS.2004.1285473]

Cui, Y., Xu, K., Xu, M.W., Wu, J.P., 2002. Stress Testing of OSPF Protocol Implementation Based on Large-scale Routing Simulation. Proc. 10th IEEE Int. Conf. on Networks, p.63-68. [doi:10.1109/ICON.2002.1033290]

Cui, Y., Xu, K., Xu, M.W., Wu, J.P., 2003. Internet Routing Emulation System and Stress Testing. Proc. 10th Int. Conf. on Telecommunications, **2**:1020-1026. [doi:10.1109/ICTEL.2003.1191578]

Elbert, M., Mpagazehe, C., Weyant, T., 1994. Stress Testing and Reliability. Conf. Record, Southcon'94, p.357-362. [doi:10.1109/SOUTHC.1994.498132]

Gullo, L.J., Davis, R.J., 2004. Accelerated Stress Testing to Detect Probabilistic Software Failures. Proc. Annual Reliability and Maintainability Symp., p.249-255. [doi:10.1109/RAMS.2004.1285456]

Hobbs, G.K., 1987. Development of Stress Screens. Proc. Annual Reliability and Maintainability Symp., p.115-118.

Hobbs, G.K., 1992. Highly Accelerated Stress Screens—HASS. National Electronic Packaging and Production Conf.-Proc. Technical Program. Cahner Exposition Group, Des Plaines, IL, USA, p.1565-1572.

Liu, L., Lin, J., Li, Z.T., Li, J.C., 2006. State Machine Based CDMA Stress Testing Service System. Asia-Pacific Services Computing Conf., South China University of Technology, Guangzhou, China, p.625-628.

McLinn, J.A., 1998. Ways to Improve the Analysis of Step-stress Testing. Proc. Annual Reliability and Maintainability Symp., p.358-364. [doi:10.1109/RAMS.1998.653804]

Pachucki, D.E., 1994. Environmental Stress Screening Experiment Using Taguchi Method. Proc. 43rd Electronic Components and Technology Conf.. IEEE Computer Society, New York, p.1028-1036.

Roy, K., Roy, R.K., Chatterjee, A., 1995. Stress Testing of Combinational VLSI Circuits Using Existing Test Sets. VLSI Technology. Proc. Technical Papers, Int. Symp. on VLSI Technology Systems and Applications, p.93-98. [doi:10.1109/VTSA.1995.524640]

Shinner, C., 1988. The board electronic STRIFE test (B.E.S.T.) program. *Am. Soc. Qual. Control Rel. Rev.*, p.3-6.