

# Automatic Stress and Load Testing for Embedded Systems

Mohamad S. Bayan   João W. Cangussu  
Department of Computer Science  
University of Texas at Dallas  
{msb021000,cangussu}@utdallas.edu

## Abstract

*Load and Stress testing are important to guarantee the system is able to support specified load conditions as well as properly recover from the excess use of resources. The generation of test cases to achieve levels of load and stress is a demanding task. Here we propose an approach for the automatic generation of test cases to achieve specified levels of load and stress for a combination of resources. The technique is based on the use of a PID Controller to drive the inputs to the desired levels. The preliminary results are a good indication of the potential of the approach.*

## 1 Introduction

Load Testing [4, 5] assesses how a system performs under a given "load." The rate at which transactions are submitted to the system is called the Load [3]. One of Load Testing objectives is to determine the maximum sustainable load the system can handle. It reveals programming errors that would not appear if the system executes with a small/limited workload. Such errors are called Load sensitive faults and emerge when the system is executed under a heavy load. Stress Testing [3, 4, 5] refers to subject a system to an unreasonable load with the intention of breaking it. A stress test denies a system the resources (e.g., RAM, disk, interrupts, etc.) needed to process a certain load. It is designed to cause a failure. It tests the system's fault recovery capability. The system is not expected to process the overload without adequate resources, but to behave (e.g., fail) in a reasonable manner (e.g., not corrupting or losing data).

In this paper we propose an automatic stress and load testing technique based on the concepts of feedback control theory. By automatically driving the resource usage to its limit, load sensitive faults can be detected and performance issues can be verified under stress conditions. Initial experiments have also shown that memory leaks can also be identified by the proposed technique. The automatic identification of these faults can have a large impact on the quality

of the products released as well as on the reduction of the required test effort.

The proposed approach is based on the application of a feedback PID (Proportional, Integral, and Derivative) controller to drive the input and make the system achieve a specified level of resource usage. For example, if the user defines the system should be tested with a memory use of 95%, starting from an initial input value; the PID controller will automatically change the input(s)/test case(s) until the desired level of stress has been achieved. Stress can also be simultaneously achieved for more than one resource of interest, for example, memory and bandwidth. Finally, experiments have shown that resources used by the PID controller itself are almost negligible and will have minor or null impact on systems resources.

Most of the work done with respect to stress and load testing are either domain specific or restricted to the stress of only one resource. Other approaches require the availability and analysis of source code limiting its scope and applicability. The approach proposed here can be potentially applied to any system and it can control any desired resource. In addition, the automation of the complete process appears to be more feasible than existing approaches.

The remainder of this paper is organized as follows. A brief description of a PID controller is presented in Section 2. The general structure of the proposed approach along with preliminary results are the subject of Section 3. Section 4 describes pertinent related work while Section 5 concludes the paper and presents future work directions.

## 2 PID Control

A PID [7, 9, 10] (Proportional-Integral-Derivative) controller is often used in closed-loop control systems. Closed-loop control system utilizes feedback based on the real-time measurement of the process being controlled. The measurements are constantly fed back to the controlling device that drives the process. The controlling device makes the measured value of the process, usually known as the PROCESS VARIABLE, follow the desired value, usually known as the

SETPOINT. Usually, this involves keep tracking of past values of both PROCESS VARIABLE and SETPOINT. Figure 1 represents a simple closed-loop control system. while Eq. 1 is the formulation of a PID control algorithm.

$$y(t) = K_P e + K_I \int_0^t e dt + K_D \frac{de}{dt} \quad (1)$$

where  $y$  is the output (CONTROL VARIABLE) of the controller at time  $t$ , and  $e$  is the error signal (ERROR) at time  $t$ .

The proportional control ( $K_P e$ ) determines the ERROR, i.e., the difference between the SETPOINT and the PROCESS VARIABLE, and then applies appropriate proportional changes to the CONTROL VARIABLE to eliminate the ERROR. The proportional control deals with the current behavior of the process.

The integral control ( $K_I \int_0^t e dt$ ) adds long term precision to a control loop and it is needed to derive the process toward the SETPOINT. It is the sum of all past ERRORS in the process over time. The integral control deals with the past behavior of the process.

The Derivative control ( $K_D \frac{de}{dt}$ ) helps in reducing overshoot and improving the response time. It monitors the rate of change of the PROCESS VARIABLE and make changes to the CONTROL VARIABLE accordingly. In other words, the Derivative control may increase the stability of the process by predicting the process behavior.

Some Systems use just PI or PD controllers for efficient control. However PID controller is preferred where efficiency, stability, and performance are required.

$K_P$ ,  $K_I$ , and  $K_D$  are user-defined parameters. They vary from one control system to another and they need to be tuned to optimize the precision of control. Determining the values of these parameters is called PID Tuning. PID Tuning objectives are system stability, quick system response, minimum oscillations, and minimum ERROR value. There are several methods available for PID Tuning, please refer to [7, 9] for detailed description of these methods.

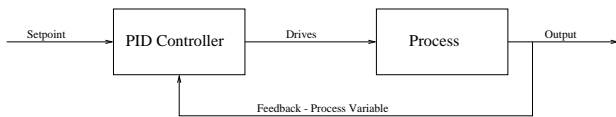


Figure 1. Structure of a PID Controller

### 3 Proposed Approach

As stated earlier, the purpose of this technique is to automate stress and load testing. Resource Saturation is a possible problem that any software system may face after running for a long time or while running with a heavy load. To

detect this problem, stress and load testing becomes essential.

The technique utilizes control theory by the use of a PID controller to derive test cases automatically. In Figure 2, PID controller accepts a setpoint as an input which represents the level of resource usage need to be achieved by the application. The PID controller will use an initial test case, defined by the tester, to drive the application to the desired level of usage, independent of the initial test case value. Every time the current usage is fed back to the PID controller, it generates a new gain. In general, Positive gain represents an increase in the input and negative gain represents a decrease in the input.

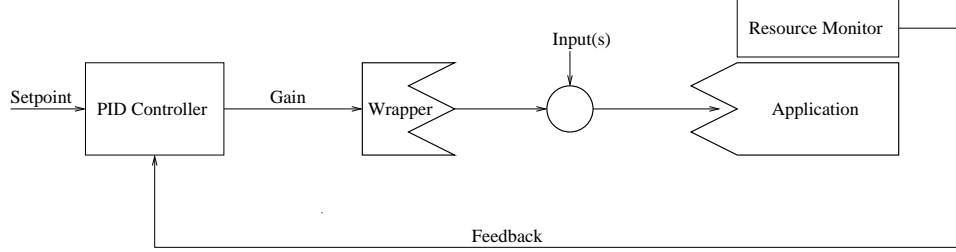
Numerical inputs can be automatically controlled by the proposed approach. However, to represent the gain in terms of non-numerical inputs, a wrapper is used. The Wrapper will utilize the gain produced by the PID controller and the previous set of input(s) to come up with the new set of input(s). For example, if the input is a string and the gain is based on the size of the string, the wrapper will create a new string based on the specified gain. Wrappers are application specifics.

In addition to the application under test, there is a resource monitor that retrieves the current usage of the resource of interest and feeds it back to the PID controller.

The approach as presented in Figure 2 appears to be very simple and it is indeed when a single variable can be controlled by a known numerical input variable as the case study presented in section 3.1. However, the majority of the applications have a large number of input parameters and their relationship with the resource of interest is not necessarily well defined. Assume an application has 10 input parameters  $P_1, P_2, \dots, P_{10}$  and let  $R$  be the resource of interest.  $R$  can be affected by any parameter  $I_j$ ,  $j = 1, 2, \dots, 10$ , or any combination of size 10 or less of them. There are a total of 1023 such combinations. Now if you assume that it takes at least three test cases to determine if a specific combination has an affect in  $R$ , a brute force technique will demand a total of 3069 test cases to properly identify the input parameters<sup>1</sup> that affect resource  $R$ . this number will go to almost 100K test cases when the number of input parameters reaches 15. Clearly, such approach is not adequate a more efficient technique needs to be designed. here we foresee the use of system identification techniques to reduce the number of test cases needed to identify resource sensitive input parameters. However, this study is beyond the scope of this paper.

Another aspect that makes the application of the proposed technique not straightforward is the simultaneous control of multiple resources. For example, one may want

<sup>1</sup>Input parameters here can be explicitly defined input variable and/or any environment variable that can be (re)defined and affects the behavior of the application.



**Figure 2. Proposed Approach**

to achieve stress levels of memory and CPU usage at the same time. However, the input variables that affect the resources may be conflicting. That is, an increase in memory usage may lead to a decrease in CPU usage or vice-versa leading the two controllers to a continuous increase/decrease of the variables without reaching the desired level of stress. Properties of input variables will have to be defined and identified to avoid such scenarios.

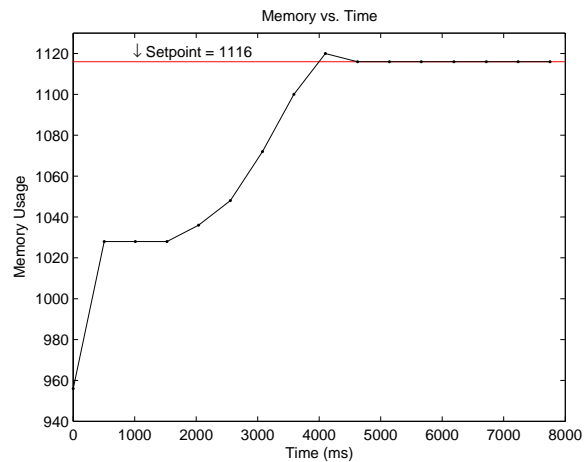
### 3.1 Preliminary Results

A preliminary experiment has been conducted using two-dimensional data multiplication to show the applicability of the proposed approach. The application used is a matrix multiplication which accepts the order  $N$  of the matrices as the input parameter. It creates two square matrices of order  $N$ , multiplies the two matrices, and stores the result in a third matrix. The resource of interest is memory; the resource monitor feeds back the memory usage by the matrix application to the PID controller after processing each test case.

The wrapper accepts the gain produced by the PID controller as an input, in addition to the last executed test case to generate a new test case. For example, in the matrix application if the last executed test case had an order equal to 10 and the PID controller produced a gain of 50% then the wrapper will generate a new test case where the order is 15. However, if the PID controller produced a gain of -30% then the new test case would have been 7. In this case the wrapper does not need to translate the meaning of a gain since the input is already a numerical value.

The application started with an initial test case of  $N = 2$  (Matrix Order) and the desired memory usage was 1116 (setpoint). As the time advanced, the PID controller produced positive gains, and the order of the matrices kept on increasing until the memory usage exceeded the setpoint to 1120 and then converged back to the setpoint of 1116. The overshoot is clear in Figure 3, where the x-axis represents the time in milliseconds, the y-axis represents the memory usage by the matrix application, and the horizontal line represents the desired memory usage. In the case where the setpoint represents the maximum load of the application, the

overshoot presented in Figure 3 could be undesired. Tuning the controller, in this case decreasing the value of the proportional gain variable  $K_P$ , can eliminate the overshoot.



**Figure 3. Results for initial test case  $N = 2$ .**

Figure 4 shows a different run of the system with an initial order input of  $N = 120$  and a setpoint of 1116. After 26 ms, the application created the matrices, multiplied them, and allocated a total memory of 1132. The memory usage oscillated around the setpoint before merging to the desired level of 1116. Figure 4 shows the changes in memory usage with respect to time and it clearly shows the oscillations of the memory usage. Again, tuning the controller (i.e., appropriately changing the values of  $K_P$ ,  $K_i$ , and  $K_D$ ) can minimize or eliminate the oscillations.

In addition to achieve automatic load and stress testing, the proposed approach helps in detecting continuous memory leak. While the resource usage is increasing, the PID controller produces positive gains, the produced test cases changes in a certain direction X. After reaching the setpoint, if the resource usage keeps on increasing, PID controller produces negative gains, and the produced test cases changes in a way opposite to direction X. This scenario is a clear indication of a memory leak. In the matrix application example, when the memory usage passes the setpoint and

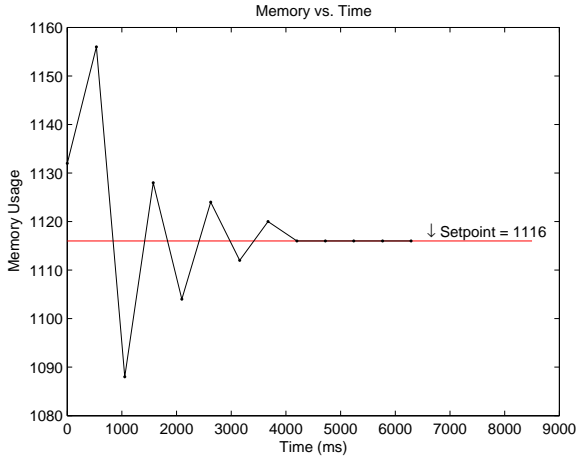


Figure 4. Results for initial test case  $N = 120$ .

keeps on increasing, the controller produces a negative gain which results in decreasing the order of the matrix until it reaches an order of zero. Exceeding the setpoint in memory usage, at the same time, where the order of the matrix was decreasing revealed the injected memory leak as seen in Figure 5.

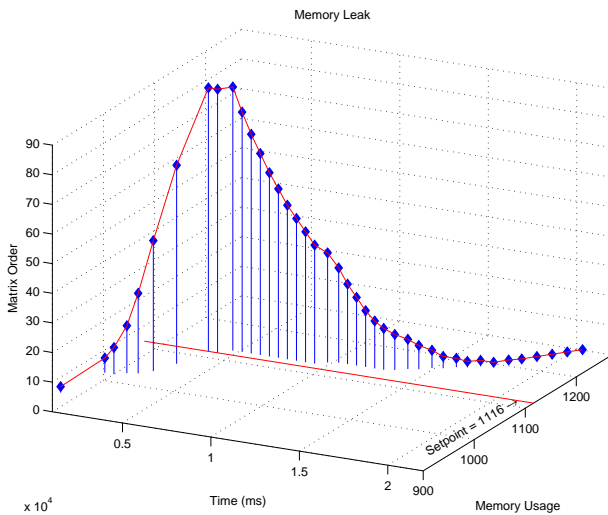


Figure 5. Memory usage with an injected memory leak for the matrix multiplication example.

## 4 Related Work

Both black box and white box approaches have been proposed for Load Testing. One example of a white box ap-

proach is provided by Yang and Pollock [11]. The technique identifies the load sensitive parts in sequential programs based on a static analysis of the code. A load sensitive part is "...one with the property that its correctness depends on the amount of input data or the length of time that the program will execute continuously." [11]. Another white box approach is Grosso's [8] Stress Testing method. It uses Genetic Algorithms to generate test cases in order to identify buffer overflow threats. In this method, static analysis and program slicing with evolutionary testing are combined to detect buffer overflow. This approach deals only with buffer overflow and not resource saturation.

Different black box testing techniques have been proposed for Load and Stress testing. One particular method is called Deterministic Markov State Testing [1, 2]. This approach is limited to applications that can be modeled by a Markov Chain since the input data is assumed to arrive according to a Poisson distribution and is serviced in an exponential distribution. The operational profile is used to build a Markov chain that represents the software's behavior. Only the most likely test cases, as computed from the most probable Markov chain states, are generated at planning time; i.e. before the start of system test. Each test case certifies a unique software state. This approach is similar to our idea where they both concentrate on resources available. However, this approach is limited to systems that can be modeled by Markov Chain and it generates the test case before starting the system test. On the other hand, our approach is more general and it generates the test cases automatically.

Briand [6] developed a method based on Genetic Algorithms to analyze real-time architectures and determine whether deadlines can be missed. The proposed method generates test cases, concentrating on seeding times for aperiodic tasks, such that completion times of a specific task's execution are as close as possible to their deadlines. Our approach is different, as our main concern is any resource saturation (including time) and Briand's method focuses on missing deadlines.

To address Stress Testing in multimedia systems, Zhang [12] developed a technique and a tool to automatically generate test cases. We share the same goal with this approach, which is resource saturation and having enough resources for the system under test with all different loads. The technique is applicable where the specifications consist of a temporal event Petri net and some temporal formulas for describing the relationships among media objects. From the Petri net, possible execution sequences are extracted. For each sequence, the event timings which maximize resource usage during a fixed finite period is computed. So, basically this algorithm divides the system statically into a set of objects where the resource usage of each object is known statically, and then simulates these objects in petri

nets to come up with different execution sequences. This approach appropriately tackles multimedia systems where the system consists of a set of media objects and where each media object is a stand alone object with known resources requirements. Not any system could be divided into such objects and not all systems could be specified using petri nets. In other words, this algorithm has applicability limitations. In comparison to our approach, our approach is applicable to any system but it requires the development of the wrapper which could pose some challenges. On the other hand, if a system could be divided into different objects with known resources requirements and these objects could be specified easily in petri nets then it's easier to use Zhang's technique.

## 5 Conclusions and Future Work

The use of a PID controller to drive inputs and achieve a desired level of load/stress for a particular resource has been explored in this paper. The preliminary results are a clear evidence of the potential of the proposed approach. In addition, memory leaks can also be identified by a proper analysis of the controller results and the resource under control.

The results presented here are based on small scale applications with a single numeric input. Case studies with higher complexity and multiple non-numeric inputs are planned to validate the applicability and accuracy of the approach. Another aspect to be considered in the future is the identification of the inputs that are sensitive to the resource(s) of interest. Finally, a general formal structure for the wrapper will be investigated.

## References

- [1] Alberto Avritzer and Brian Larson. Load testing software using deterministic state testing. In *Proceeding of the International Symposium on Software Testing and Analysis*, pages 82–88. ACM, June 1993.
- [2] Alberto Avritzer and Elaine J. Weyuker. Generating test suites for software load testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 44–57, Washington, Seattle, August 1994. ACM.
- [3] Boris Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, 1984.
- [4] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 2nd Edition, 1990.
- [5] Robert V. Binder. *Testing Object-Oriented Systems - Models, Patterns, and Tools*. Addison-Wesley, 1999.
- [6] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. Stress testing real-time systems with genetic algorithms. In *Proceedings of the Genetic And Evolutionary Computation Conference*, pages 1021–1028, Washington, DC, June 2005. ACM.
- [7] Graham C. Goodwin, Stefan F. Graebe, and Mario E. Salgado. *Control System Design*. Prentice Hall, 2001.
- [8] Concettina Del Grosso, Giuliano Antoniol, Massimiliano Di Penta, Philippe Galinier, and Ettore Merlo. Improving network applications security: a new heuristic to generate stress testing data. In *Proceedings of the Genetic and evolutionary computation Conference*, pages 1037–1043, Washington, DC, 2005. ACM.
- [9] John A. Shaw. Pid algorithms and tuning methods. <http://www.jashaw.com/pid/tutorial/>.
- [10] T. Wescott. Pid without a phd. Technical report, Embedded System Programming, 2000.
- [11] Cheer-Sun D. Yang and Lori L. Pollock. Towards a structural load testing tool. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 201–228, San Diego, CA, 1996.
- [12] Jian Zhang and S. C. Cheung. Automated test case generation for the stress testing of multimedia systems. *Softw., Pract. Exper.*, 32(15):1411–1435, 2002.