A variant of this paper appeared in Proc. First Workshop on Hot Topics in Autonomous Computing (HotAC 06), June 2006, pp.17-21.

Software Error Early Detection System Based on Run-time Statistical Analysis of Function Return Values

Alex Depoutovitch and Michael Stumm
Department of Computer Science and Department Electrical and Computer Engineering
University of Toronto
{depout,stumm}@eecg.utoronto.ca

Abstract

Large software systems are extremely complex and based on code that is constantly changing with bug fixes and new features. As a result, these systems will likely never be free of bugs. The bugs typically don't expose themselves until they are triggered by a new workload, and when triggered, they are rarely immediately fatal, but result in a system that continues to with corrupt internal state, deteriorating over time to the point where it becomes inoperable. Having a method to identify corrupt state early would allow the initiation of defensive actions such as flushing page caches or redirecting external requests to another service in the cluster.

In this paper, we propose a statistical method of detecting problems in software at run-time based on analyzing function return values. The methodology, at this time, requires the availability of source code, but does not require understanding the source code. Our experimental results indicate that our method can be effective in identifying problems early on, potentially allowing for defensive measures. The overhead is negligible at less than 1%.

1 Introduction

Large software systems, such as operating systems or databases, are extremely complex, with internal state defined by many thousands of parameters. These systems can be in any one of a very large number of states at any given time. Moreover, these systems tend to be in a constant flux with frequent bug fixes and addition of new features, so it is impossible to fully test these systems or predict how they will behave precisely in future scenarios, and it is unlikely these types of systems will ever be entirely free of bugs.

Operating system kernels are a good example of such a complex software system. An operating system has thousands of internal functions that interact with each other and with the outside world, and it has thousands of data structures maintaining the internal state of the system. The system must be preemptible, be able to run concurrently on multiple processors sharing state, and must scale reasonably well. In addition, a modern operating systems typically contains third-party extension modules that are loaded into the kernel dynamically at the run-time and that interact with the rest of the operating system. Often, those writing a kernel component or an extension use only a small part of published interface and do not fully understand how other parts of the system work internally or interact with each other. Hence, operating systems will likely always have bugs.

Our goal is to measure the general well-being of a target software system and assess the likelihood of a pending failure at run-time. Our approach is inspired by other areas of science, such as thermodynamics or economics, that use statistical methods to describe complex systems. They typically define a small set of global parameters (e.g., temperature or key macro-economic indexes) which are derived from many micro-parameters (e.g., velocities of all molecules or multiple detailed economic indicators) using averages or other more elaborate statistical functions. Then, an approximate model of the system is defined based on the global parameters and a set of rules that describe expected relationships and behavior.

In the remainder of the paper, in Section 2 we first describe related approaches and then in Section 3 give an overview of the general framework we. In Section 4, we propose a specific approach based on monitoring function return values in real-time, identifying periods when the percentage of error return codes exceeds a threshold. Our implementation is described in Section 5, and in Section 6, we present preliminary results of our experiments that show that our method can be effective in identifying problems at an early stage while imposing minimal overhead.

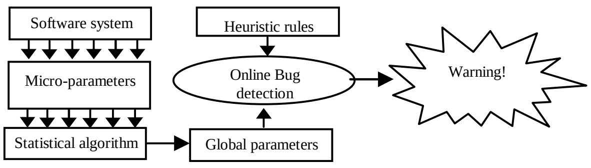

Figure 1. The general framework

2 Related Work

A number of groups have applied statistical methods to predict pending software faults, to assist in identifying the existence of bugs, and to detect sub-optimal operating conditions. Goldszmidt et al. published a nice article summarizing common problems in applying machine learning and statistical methods in systems research [1]. They show the benefits of statistical analysis and machine learning, such as the ability to automatically adapt algorithms to system and environmental changes.

Gross et al. give examples of software aging problems, where an application can work well for some (often very long) time but then requires a restart [2]. The authors suggest using statistical pattern recognition to predict the time when a restart is required. Their approach has two key disadvantages: (i) it requires detailed knowledge of the software, and (ii) the method must be adapted whenever the software is modified.

Our approach, that will be discussed in next section is similar to the one implemented by S. Hagal et al. in DIDUCE tool [3]. In order to detect abnormal software behavior, they monitor values of all class members and global variables defined in the application. However, this approach introduces 1-2 orders of magnitude run-time overhead and makes it unsuitable for monitoring production system or detecting bugs that occur infrequently.

E. Kiciman et al. use unusual run-time component interaction patterns to detect anomalies in a running program [4]. Their tool uses a modified middleware server to trace inter-component calls and a decision tree algorithm to detect unusual patterns. Similarly, G. Jiang et al. monitor run-time execution paths and uses n-grams and automata to detect anomalies [5]. All of these tools were implemented for programs running in Java–based middleware environment and are not capable of predicting failures in standalone software compiled into native code, i.e. C++ applications. Having such capability, however, is desirable, and feasible because, as Hennessy noted [6], catastrophic failures rarely occur in real systems without being preceded by many smaller non-fatal errors. Gradual failures are often not visible, because the software tends to ignore them, work around them, or correct them.

3 General Framework

The methodology we propose to detect abnormal software system state falls under the general framework depicted in Figure 1. *Micro-parameters* that describe many tiny aspects of the system, such as function return values or the time spent waiting for a lock, are monitored and collected at run-time. However, each of these micro-parameters may not be very meaningful on their own, and the amount of data generated will be too voluminous for direct consumption. For this reason, statistical methods can be used to process the large amount of fine-grained data, filtering out irrelevant noise to produce more meaningful *global parameters*. With an appropriate set of global parameters, it is possible to define acceptable ranges for their values, as well as rules as to how the global parameters are expected to relate to one another. System state can then be viewed as having been corrupted if global parameter values lie outside the acceptable ranges or if the rules are violated.

With such a framework, four questions need to be addressed:

1. Which micro-parameters can and should be monitored in a system?
2. What global parameters can be defined that can be effectively calculated from the micro-parameters and are meaningful at the same time?
3. How do the defined global parameters relate to each other and what ranges are acceptable for their values?
4. How can these relationships be used to discover bugs, predict system failures, and measure the general well-being of the system?

Collecting the right set of micro-parameters is the most critical step because they provide the foundation for all subsequent statistical calculations. The following list contains examples of the micro-parameters that are well suited for describing system state:

- commonly used performance metrics such as: CPU, memory and I/O load introduced by specific parts of the system, various queue lengths, cache miss rates, and data from various hardware counters;
- size of data allocated; e.g., it may be number of instantiated objects of each type;
- error values returned by individual functions;
- the time it takes to execute each function.

These parameters have the property that they are applicable to all parts of the software system and require minimal knowledge of the specifics of the software. For example, for function return values, we only need to know what return values indicate a fault. Very often there are known specific values that indicate an error, or in case of functions returning a pointer, a null pointer typically indicates a fault. More importantly, what constitutes an error return value can be determined automatically using statistical methods.

In this general framework, code is injected into the software to monitor and collect micro-parameters, and to periodically invoke a statistical engine. The statistical engine processes the micro-parameters to obtain values for global parameters and then to (i) identify violated rules, and (ii) identify parameter values that lie outside acceptable ranges.

To automatically identify acceptable ranges for the global parameters, it is possible to generate the global parameters on a running system assumed to operate correctly with a reference workload, recording the ranges encountered. Then, when the target software is run with new workloads, parameter values that lie outside the ranges encountered with the reference workload may indicate an abnormal situation.

4 Analyzing Return Values

The particular method we propose uses the above framework based on monitoring function return values. Specifically, we attempt to monitor the rate at which functions return an error value. To determine which function return values represent an error, we initially assume that error is indicated with 0 for functions returning pointers and 0 or –1 for all other functions, and we validate this assumption on each function individually by running a *reference load* that exercises the full functionality of the system. We refer to the recorded data *as reference results*. Running the reference load allows us to determine the frequency at which each function is expected to return what is assumed to be an error value. Experimentally, we have found that the reference results are not sensitive to the specific set of applications we use as a load, as long as the load exercises most of the system functionality. However, to obtain meaningful reference results, the system from which reference results are obtained must be stable.

After producing the reference results, we can then run the system under real workloads and measure the number of error values returned by its functions relative to the reference results. The absolute number will, of course, depend on the particular system workload, so normalization is required. We normalize by dividing by the total number of function calls (i.e. percent of functions returning error codes).

More precisely, the global parameter we monitor is the difference between the number of functions returning errors and the same number calculated from the reference results, normalized by total number of function calls. In an ideal situation, where the reference load is representative of future workloads, we expect this parameter to be close to zero if the software has not encountered any bugs. We have found that it is legitimate for this parameter to be slightly greater than zero (~0.01%), i.e. a helper function that allocates a chunk of memory from a heap may return an error when the current heap size is not enough to accommodate an incoming request and needs to be increased. On the other hand, if the state of the system has become corrupted, then we expect the parameter to significantly deviate from zero.

The advantage of the proposed approach is that it can be applied to any large software system without understanding the code base and without knowing which function return values indicate error.

5 Implementation

We have applied the techniques described in the previous section to the K42 open source operating system [7]. K42 is written mostly in C++. We selected K42 for our experiments in part because it is a system still under active development that we understand well and for which we have bugs we can easily inject. However, we envision using K42's hot-swapping capability to replace objects at run-time when the early warning system identifies a problem.

We implemented a C++ preprocessor that automatically scans C++ source files, finds functions that return either system status code or a pointer. K42 contains approximately 4,500 functions. Roughly 1,800 of them return either a system status code or a pointer and we only record the return values of these functions. (System status code values are usually the same throughout the system.) The preprocessor injects code that records the return value along with the address of the corresponding return statement. The return address will help us identify functions that returned error values and execution paths in offline analysis. The injected code also increments a counter of function calls, and once in every N calls, triggers the statistical analysis engine to process and analyze the collected data.

The statistical engine periodically computes the global parameter from the collected set of micro-parameters and analyzes it in the hope of detecting when the system enters a state that might be of interest. Specifically, the engine considers the system to have gotten into an abnormal state when the normalized number of functions returning an error value exceeds some threshold. This analysis must be done frequently

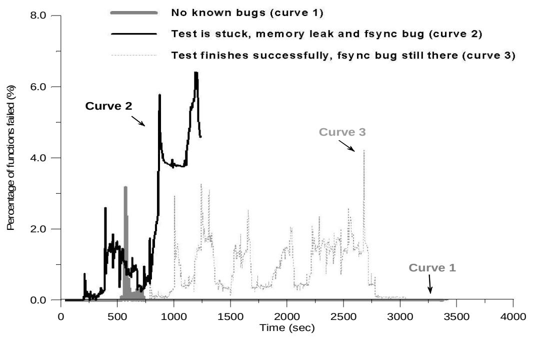

Figure 2. Percentage of function failed during MySQL table creation benchmark

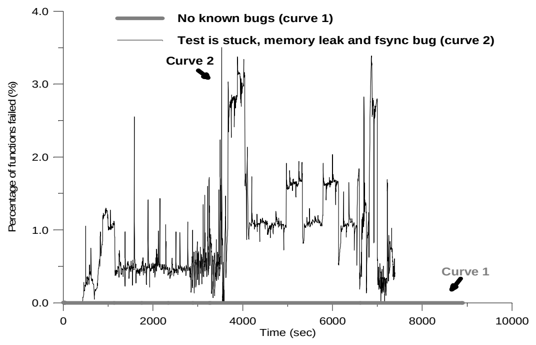

Figure 3. Percentage of function failed during MySQL data insertion benchmark

enough so that the system entering an abnormal state can be detected soon after the state becomes corrupted in order to have time be able to react to it and possibly take corrective or recovery actions. At the same time, the analysis cannot be too frequent so as not to introduce too much overhead.

6 Experimental Results

In this section, we present our findings from experiments using the K42 operating system running the MySQL database server with a benchmark load. The benchmark suite we used was created by the MySQL team to test the performance of MySQL server on different platforms [8]. We ran a number of tests from this suite using the K42 operating system with known bugs to determine (i) whether we could detect the existence of the bugs in the system by monitoring the global parameter we defined, and (ii) whether the resulting abnormal system state could be detected early enough to provide sufficient time for corrective actions. With a positive answer to both of these questions, several attractive applications are possible as described in section 7.

For our experiments, we modified the K42 kernel by adding the statistical analysis engine, and used the preprocessor to inject the code to collect the function return values. The reference load we used was the full set of regression tests built up by the K42 team over the years, and when run, we made sure these tests produce the expected results. We ran the same workload on the system with and without the instrumentation. The total overhead introduced by both the injected code and the statistical engine was less than 1%, which can be considered negligible.

After running the reference load, we ran the MySQL load with two real bugs. One of the bugs, a resource leakage in the file access code, was discovered beforehand and was reintroduced in order to test our approach. The second one, an I/O synchronization problem, was discovered accidentally by our approach and we had been unaware of it before we ran our experiments.

While running the system, we collected the statistics in real time until the system or application crashed or froze. After that, we determined whether the difference between the global parameter values observed in the system with and without the bugs was large enough to be noticeable and measured the time period between the point where the abnormal situation is clearly identifiable for the first time and the time the system crashed or froze.

Figure 2 depicts the percentage of functions that return error values over the total number of function calls. The workload that was used to produce these curves was the table creation test from the MySQL benchmark suite. The figure contains three curves for three different system runs we measured. The first run (curve 1) used the kernel that did not have any known bugs. The error rate is around 0.01% with small spikes up to 3.5%. The spikes, however, are very short, for periods of at most 0.1-0.5 seconds and a simple criteria based on minimal duration of abnormally high error rate can filter such false positive events out.

For the second run (curve 2), the kernel contained both bugs described above. The second curve ends at the point where the benchmark test stopped running because the system ran out of resources. The percentage of the function calls returning error values starts to increase dramatically at approximately 400 seconds after the start of the benchmark. From that point on, the percentage of functions calls returning error values stays at 1-1.5% for another 300 seconds and after that increases even further to 4-7%, remaining high until the system fails.

For the third run, we used the kernel with the resource leakage bug fixed, but with the I/O bug remaining. In this case, the benchmark test completes successfully, but our statistical analysis shows that there are still some problems. The percentage of functions calls returning error values is consistently larger than zero and stays at 0.5-2% with spikes up to 4.5% until the benchmark finishes. This made us suspect there was another bug. After carefully analyzing which functions return error values and inspecting the code, the second bug in the I/O synchronization routine was found.

The third run demonstrates, in our view, that the methodology we are proposing can be used in a real life situation to detect and identify bugs even if regression tests pass successfully. As we can see from the curves, with the bug in the system, the percentage of the functions returning error values is abnormally high (i.e. 0.5-2%) for a prolonged period of time (i.e. thousands of seconds). This is sufficient to react, should we decide to do so.

Figure 3 depicts the results of another benchmark, namely the data insertion benchmark from the same MySQL benchmark suite. Again, running the kernel with the resource leakage bug causes the system to run out of resources and freeze before the benchmark ends, but the synchronization bug does not reveal itself, since the code that contains the bug is not exercised. Because of this, only two curves are presented. We can see again that when there is a problem in the operating system kernel (curve 2), our analysis discovers it well in advance. The percentage measured is around 1-1.5% for long periods of times (i.e. hundreds of seconds) with spikes up to 3.5%. In contrast, the curve for the run with the bug-free version of the kernel (curve 1) is difficult to see because the error rate very close to zero; i.e., constantly less then 0.01%.

7 Possible Applications

We see two main areas where our proposed method may be used: for regression testing of the system in development and for the run-time monitoring of large mission critical applications.

As we have shown, it is possible that even regression testing may leave a serious bug undetected since regression tests only consider output. Our statistical method can be used to detect bugs that don't affect the output of the program, for example when load on the system is not heavy or the run is not long enough. This allows earlier detection of newly introduced bugs.

For mission critical systems, our method can detect abnormal operating conditions, giving an early warning signal to the operator by putting the system into a "high-alert" state until the global parameters return back to normal. An early warning system also allows autonomic remedial actions such as disabling write-back caches so that all disk writes occur immediately, minimizing the amount of lost data, or automatically redirecting user requests to another computer in a fail-safe cluster and rebooting the system in abnormal state. As shown by Qin et al., rebooting the system and re-executing a request in a different environment is often enough to solve many problems without dropping the request [9].

8 Concluding Remarks

We described a new approach for detecting software problems at run time by statistically analyzing function return values. While our work is still at an early stage, initial experiments using the K42 operating system show promise. Our method was able to detect abnormal states early, and it incurs only a negligible amount of overhead. We intend to do much more extensive experimentation with other workloads and bugs. Moreover we intend to expand our method to include the analysis of additional parameters and rules that define their relationship.

9 References

[1] M. Goldszmidt, I. Cohen, A. Fox, S. Zhang. Three research challenges at the intersection of machine learning, statistical induction, and systems. In Proc. 10th Workshop on Hot Topics in Operating Systems, 2005.

[2] K. C. Gross, V. Bhardwaj, R. L. Bickford Proactive Detection of Software Aging Mechanisms in Performance-Critical Computers. 7th Annual IEEE/NASA Software Engineering Symposium, 2002.

[3] S. Hangal, M. Lam. Tracking down software bugs using automatic anomaly detection. In Proc. 24th International Conference on Software Engineering, 2002

[4] E. Kiciman, A. Fox. Detecting application-level failures in component-based Internet services. IEEE Transactions on Neural Networks, September 2005

[5] G. Jiang, H. C. Ungureanu, K. Yoshihira. Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata. In Proc. 2nd International Conference on Autonomic Computing

[6] J. Hennessy. The Future of Systems Research. IEEE Computer, pages 27--33, August 1999.

[7] J. Appavoo, M. Auslander, M. Burtico, D. Da Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, J. Xenidis. K42: an Open-Source Linux-Compatible Scalable Operating System Kernel. IBM Systems Journal pp. 427-440 Vol. 44, No. 2, 2005

[8] The MySQL Benchmark Suite http://dev.mysql.com/doc/refman/5.1/en/

[9] F. Qin, J. Tucek, J. Sundaresan, Y. Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In Proc. 20th Symposium on Operating Systems Principles, 2000