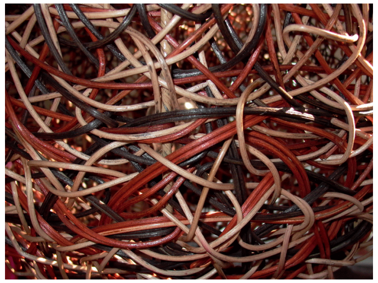
Software Error Early Detection System Based on Run-time Statistical Analysis of Function Return Values

Alex Depoutovitch and Michael Stumm

University of Toronto



System Software is Complex

- Thousands small interacting parts
- State defined by thousands of parameters
- Runs on multiple processors concurrently
- Subject to constant changes
- Different parts are written by different people
- Third party modules

System Software will always contain Bugs

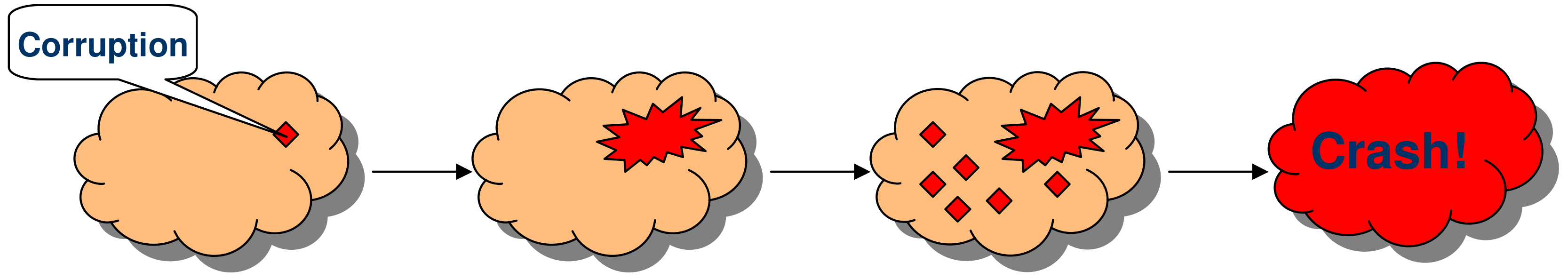
- Bugs often result in catastrophic system failure
- It is easy to notice when bug crashes the system, but usually it is too late to do anything

But …..

System crashes are often gradual

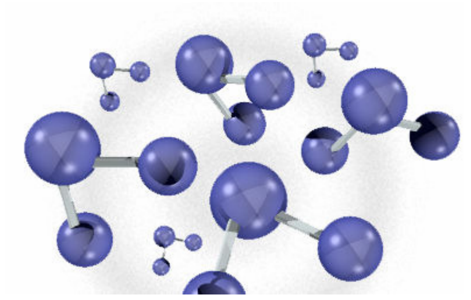
Catastrophic failures rarely occur in real systems without being preceded by many smaller non-fatal errors.

Hennessy. The Future of Systems Research. IEEE Computer



Objectives

- Continuously measure general well-being of system software
- Assess likelihood of pending failure at run-time
- Allow system to take defensive measures when in abnormal state



Similar problems

- Thermodynamics:
  - Needs to describe behaviour of 10^{23} molecules
  - Each has its own mass, velocity
  - Solves it by introducing few macro parameters: Temperature, Pressure, Volume
- Medicine:
  - Needs to describe system composed of billions of cells
  - Disease development is often hidden and gradual
  - Disease often detected by simple temperature parameter

Possible applications

- Run-time monitoring of large applications
  - Alert system administrator
  - Autonomic remedial actions (e.g., disable write-back caches)
  - Automatically redirect user requests to another computer in a fail-safe cluster and reboot system in abnormal state
- Regression testing of the system
  - Can be used to detect corrupt state even if system produces correct output

Simple Idea: monitor function return values

- Instrument code to monitor return values of functions
- Identify error return values
- At run-time statistically analyze normalized frequency of function error return values
- Compare against expected frequency from reference workload
- If higher than expected, issue "abnormal state" warning

$$y = Y_0(x)$$

Identification of error values

- Initially assume that all NULLs instead of pointer and system error codes indicate an error
- Run reference load that exercises the full functionality of the system
- Determine the reference results:
  frequency at which each function is expected to return what is assumed to be an error value
- Normalize results by dividing by the total number of function calls (i.e., expected percent of errors for each function)

Reference load

- Able to run without system crash
- Produces correct results
- System must be able to run and produce correct results with load
  - several times longer
  - several times more intensive

Global parameter

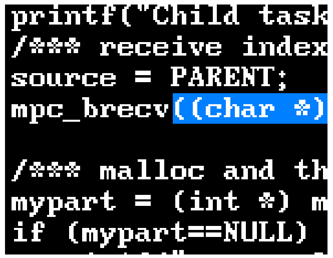
- Difference between the percentage of errors returned by functions and the same number calculated from the reference results, normalized by total number of function calls

- Parameter must be close to zero when system is in “normal” state
  In reality we found that to be around 0.01%

Tested Software System

We applied our methodology to the K42 operating system:

- Open source
- It is under active development but is also mature enough to run complex benchmarks
- We have a good knowledge of it
- It has fixed bugs that can be reintroduced
- We envision using its hot-swapping capability to replace objects at run time



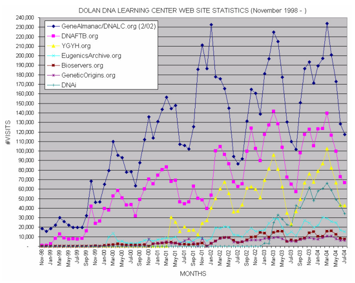
```
printf("Child task
/*** receive index
source = PARENT;
mpc_brecv((char *)

/*** malloc and th
mypart = (int *) m
if (mypart==NULL)
```

Code instrumentation

C++ preprocessor that finds functions that return pointers or system status code

- Total kernel contains 4,500 functions
- Instrumented 1,800 functions
- Instrumented code records returned value and address of return statement
- Injects code to trigger analysis engine



Analysis engine

- Periodically computes the global parameter from the function return values
- Analyzes it in the hope of detecting when threshold is exceeded
- Frequency of invocation is compromise between early detection and performance impact

In our implementation it was invoked every 1000 calls and performance overhead was less than 1%

Experiments

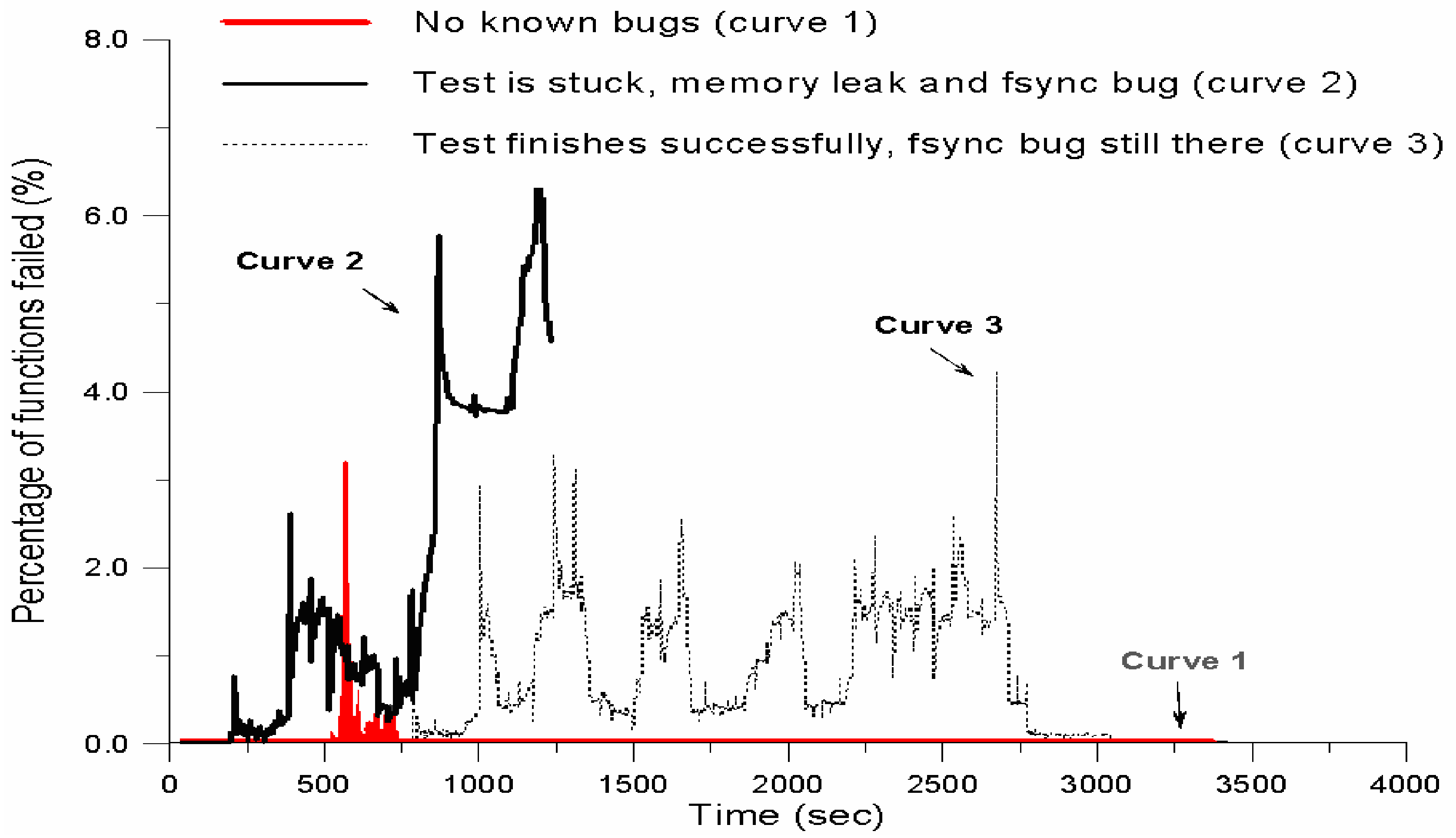
- Modified K42 operating system with injected code to collect the function return values and analysis engine
- As a reference load we used set of regression tests created by K42 team for testing K42
- MySQL 5.0 database server
- Benchmark suite created by MySQL team to test performance of MySQL on different platforms

Bugs used for experiments

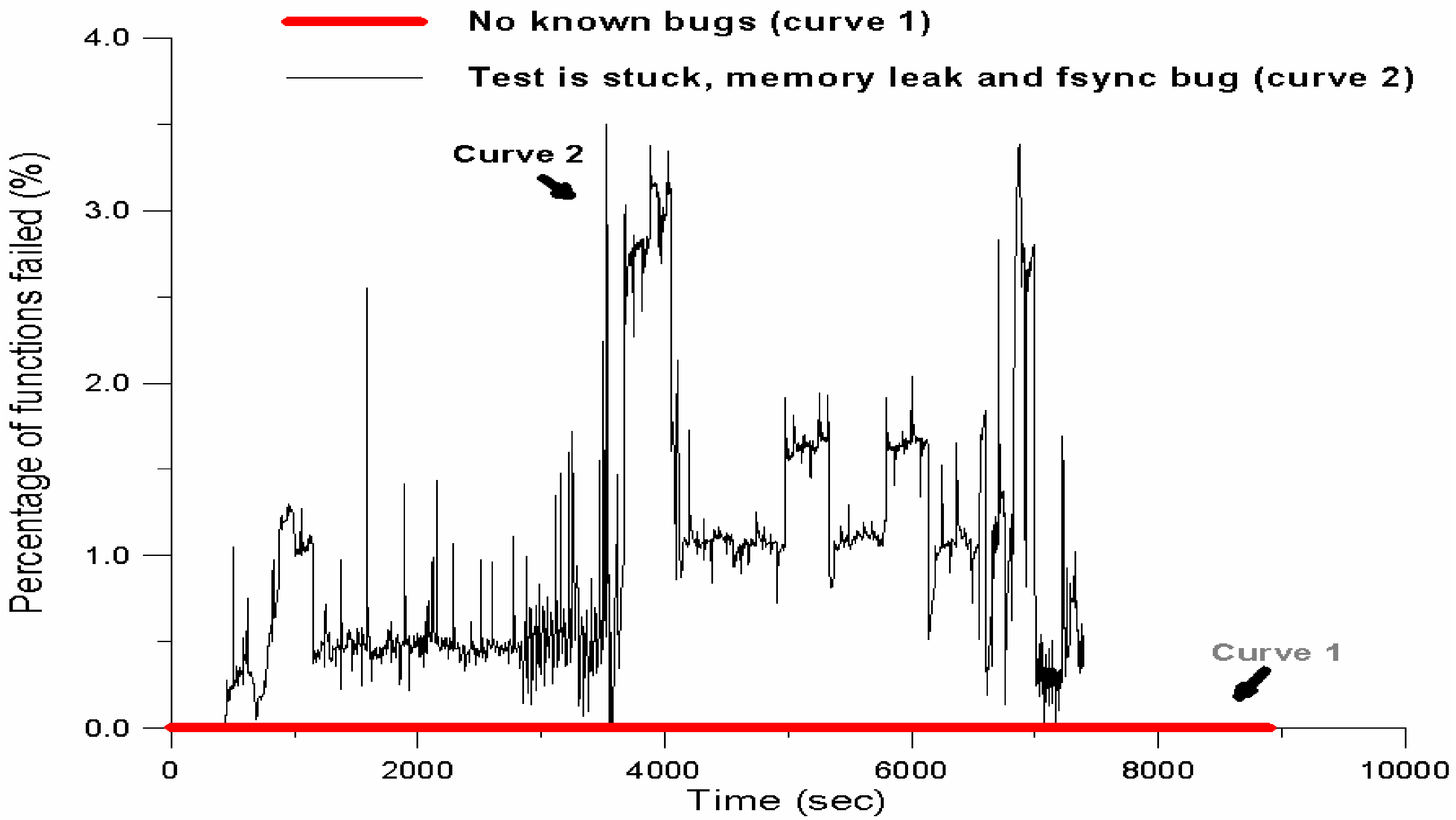
We tested our system with two real bugs:

- Bug 1:
  - Resource leakage in file access code
  - Was discovered beforehand and reintroduced in order to test our approach
- Bug 2:
  - I/O synchronisation problem
  - We did not know about it beforehand and it was discovered by our approach

MySQL table creation benchmark



MySQL data insertion benchmark

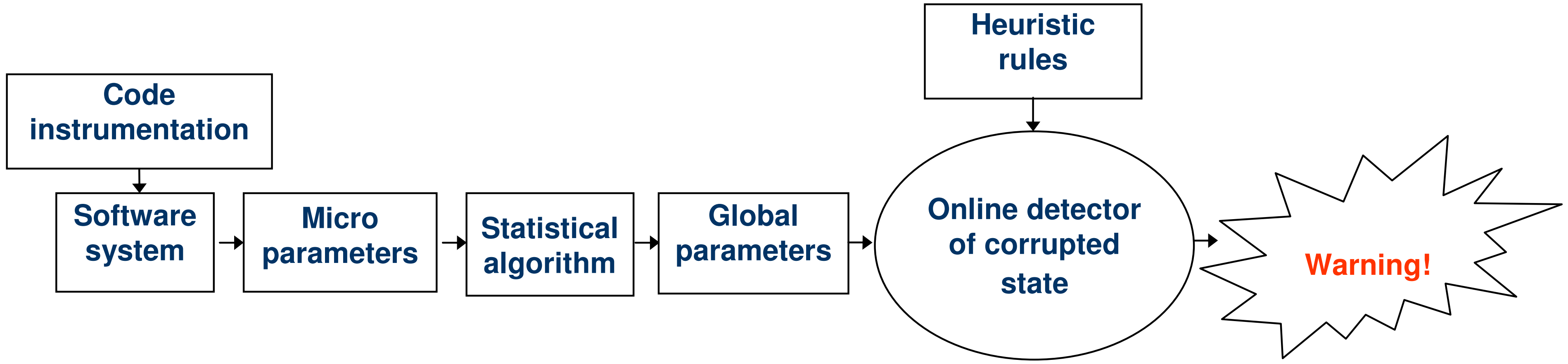


Limitations

- Apache workload
  success
- Injected bugs that skip resources locking failure
- Injected stack corruption
  Only 10% of crashes were detected

Goal: improve reliability of Early Detection System by monitoring and statistically analyzing additional parameters

General Framework



- Micro-parameters describe many tiny aspects of the system
  Are not meaningful on their own and be too voluminous for direct consumption
- Global parameters are derived from micro-parameters and describe system state as a whole
  System state can be viewed as having been corrupted if global parameter values lie outside the acceptable ranges or if the rules are violated

Constraints on micro-parameters

- Fast collection and processing
  - Less then several percents of overhead
- Generic
  - Do not require knowledge of specific system part
- Cover all parts of the system
  - Detect bug in any module

Examples of micro-parameters

- Commonly used performance metrics such as CPU, memory and I/O load, cache miss rates, data from hardware counters
- Size of data allocated; e.g., the number of instantiated objects of each type
- Error values returned by individual functions
- The time it takes to execute each function

Constraints on global parameters

- Should be effectively calculated from the micro-parameters and be meaningful at the same time
- Determine relation to each other and acceptable ranges for their values

Conclusion

While our system is on early stage of development it demonstrated following features:

- Detects previously unknown corruptions of system state
- Gives plenty of time between raising “red flag” and crash of the system
- Has low overhead, which makes it possible to use it in production systems

Questions