

Debugging, Advanced Debugging and Runtime Analysis

Salim Istaq

Computer Engineering, Deptt. of Electrical Engineering
Al-Margeb University
Al-Khoms, Libya

Aufaq Zargar

Deptt. of Computer Science
Al-Margeb University
Al-Khoms, Libya

Abstract—This paper discusses debugging and runtime analysis of software and outlines its enormous benefits to software developers and testers. A debugger is usually quite helpful in tracking down many logic problems. However, even with the most advanced debugger at your disposal, it doesn't guarantee that it will be a straightforward task to rid your program of bugs. Debugging techniques might help you in your task of flushing errors out of your program. Some of these will directly involve the debugger but many of them won't. The hope is to add to your debugging repertoire in order to assist your personal debugging quests when things go strangely wrong. Testing is more than just debugging. Testing is not only used to locate defects and correct them. It is also used in validation, verification process, and reliability measurement. Testing is expensive. Automation is a good way to cut down cost and time. When we write software applications, we need to debug them. Software Testing provides an objective, independent view of the software to allow the business to appreciate and understand the risks at implementation of the software. A primary purpose for testing is to detect software failures so that defects may be uncovered and corrected. This is a non-trivial pursuit. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions. Program testing and fault detection can be aided significantly by testing tools and debuggers. Testing and debug tools include features such as program monitors, permitting full or partial monitoring of program code including instruction set simulator, permitting complete instruction level monitoring and trace facilities, program animation, permitting step-by-step execution and conditional breakpoint at source level or in machine code, code coverage reports, formatted dump or symbolic debugging tools allowing inspection of program variables on error or at chosen points. Automated functional GUI testing tools are used to repeat system-level tests through the GUI, benchmarks, allowing run-time performance comparisons to be made, performance analysis that can help to highlight hot spots and resource usage. A runtime tool will allow you to examine the application internals after the run via the recorded runtime analysis data. Runtime analysis removes the guesswork from debugging.

It helps to uncover Memory corruption detection, Memory leak detection etc. Runtime analysis is an effort aimed at understanding software component behavior by using data collection during the execution of the component. Runtime analysis is a topic of great interest in Computer Science. A program can take seconds, hours or even years to finish executing, depending on various parameters.

Keywords-Debugging, Runtime Analysis, Code Coverage, Memory Leaks, GDB.

I. INTRODUCTION

Black box testing [9] treats the software as a "black box" without any knowledge of internal implementation. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix, exploratory testing and specification-based testing. Specification-based testing aims to test the functionality of software according to the applicable requirements [1] Thus, the tester inputs data into, and only sees the output from, the test object. This level of testing usually requires thorough test cases to be provided to the tester, who then can simply verify that for a given input, the output value (or behavior), either "is" or "is not" the same as the expected value specified in the test case. Specification-based testing is necessary, but it is insufficient to guard against certain risks [2]. The black box tester has no "bonds" with the code, and a tester's perception is very simple: a code must have bugs. Therefore, black box testing has the advantage of "an unaffiliated opinion," on the one hand, and the disadvantage of "blind exploring," on the other [3].

II. RUNTIME ANALYSIS

Many bugs cannot be determined at compile time. Run-time tools are required to find these bugs. Run-time analysis tools work at run-time instead of compile time. It provides valuable information about how the developed component or the whole application behaves when it runs, either in the test environment or in the final deployment environment. Runtime analysis provides understandings of the many aspects of application execution. Some of them are listed below:

- Code paths
- Code coverage
- Utilization of Memory
- Memory leaks and memory corruption
- Performance bottlenecks

III. DEBUGGING

Debugging is the least predictable software development activity. One usually doesn't know in advance whether finding the next bug will take five minutes or five weeks. Some people debug using their intuition. From the nature of the symptom, they can go directly to the parts of the program that are most likely to contain the error. While such intuition is useful and can be developed if you learn from experience, it is not necessary; there is a general approach to debugging that does not need it. When you run a buggy program on some test data, initially everything is all right (variables have the right values, the right output is produced and control takes the expected path). Things go wrong after a point in time at which a buggy statement is executed. One can identify the buggy statement by identifying this point in time. This of course requires the programmer to be able to tell what the right values of variables are at every point in the program. For programmers other than the original authors, and sometimes even for them, this requires documentation of what the code is meant to do. If this is missing, it has to be recreated. When writing code, we test the basic functionality of a component first and make sure that all requirements are satisfied. Later on in the development cycle, QA (Quality Assurance) teams usually test the software [10], [11]. Such QA tests often focus on the functionality of main use-case scenarios. So, if the functionality of all major use-case scenarios is confirmed, then is the application ready to be shipped to the customer? The answer is still no. The application might still crash on some machines, in some combination of scenarios, or in some untested scenarios and its performance might not be satisfactory. The task of all development roles should be to minimize the probability of shipping faulty code to the customer, and the best way to do this is to perform tests as early in the development cycle as possible. It is helpful to think of runtime analysis as an extension of standard debugging tools and methods that can help teams uncover peculiar and sometimes very difficult to resolve problems.

A. Implementation and Debugging

One can argue about the best way to develop software, but we think, we can all agree that a methodical approach is more likely to deliver high quality results than an ad hoc approach without planning or role assignments. And whether you design first and then implement features, write tests before working

on the code, or even skip process steps and just start with code, the final result the developed component or application has to be tested for functionality. It should also be associated with requirements to ensure that the final product matches users' needs. And sooner rather than later, it will require debugging. If you have ever developed software, you know that it can easily run off course. To deliver reliable software, you need to understand exactly how the application executes. This understanding should encompass not only the application's logic, but also its performance and memory considerations. At some point in the development lifecycle, the projected features need to be implemented into code; this code, in turn, needs to be compiled and linked into either a component that will be executed with the help of unit testing, or a debug version of the standalone application. Start with basic functionality that works, and start adding features in order to avoid functionality flaws later in the process. The same applies for performance and memory utilization problems: The earlier you detect performance and memory bottlenecks, the easier it will be to fix them and to deliver software of higher quality. If you write tests even before you implement the code, you can define verification points not just for the component's functionality, but also for its performance and memory usage. This is where runtime analysis comes into play. Since you have to test the functionality anyway, runtime analysis will provide you exact information about the root cause of the problem, based on the information collected during functionality testing. All major programming languages host features that provide collections of additional information about the application's execution. Using programming language features such as *assert* and *trace*, and keywords for exception handling, can help inform you about what has happened during application execution. The system APIs for timing can also help you gain information about performance, but as a data collection vehicle they can quickly become very ineffective, and they influence the performance of the tested application too much to be reliable.

B. Assert

The *assert* is used (with a Boolean-expression parameter) to check assumptions. If the expression is TRUE nothing happens, if FALSE, a message is printed and the program crashes. *Assert* works well in finding bugs early, when testing in the host environment on failure, *assert* causes a return to the host operating systems. The advantage of this debugging method is obvious, but there are some disadvantages as well:

- You can't *assert* every single condition in your code. That would make the code extremely slow and difficult to maintain.
- Numerous other errors can occur without any visible effects on the code execution; even if you combine all of the language capabilities we've mentioned, it may not be enough.

- Sometimes the root cause of the error occurs much earlier in the application execution, and it is difficult to trace back to the problem.

As we mentioned earlier, it is also possible to measure application performance from within the code. Here is an example:

```
time=System.currentTimeMillis();  
Do Something();  
time = System.currentTimeMillis() - time;  
System.out.println("Measured time is " + time);
```

However, this profiling method doesn't allow you to profile the whole application and still discover details about each method, not to mention specific lines of code. The collected time is also not reliable because it may be influenced by other processes running on the same machine user interaction and so forth. For more detailed, reliable performance profiling, you need a specialized performance profiling tool. "A software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables. A debugger allows you to stop the execution of an application at virtually any line of code. However, the debugger will not tell you whether you have a memory or performance problem. It will assist you in finding one if you have a hunch that it exists.

C. GDB

GDB, the GNU Project debugger offers you the possibility to check what happens inside a certain program when it is being executed or what a program was doing at the moment when it crashed. GDB [4] can help you in four ways in programs:

- It can start your program and specify the circumstances that can influence your program.
- It can stop your program under given conditions.
- It can investigate what happened at the moment your program was terminated.
- Make changes in your program, which allows you to experiment with correcting the influence of the bug.
- The program to be debugged can be written in C, C++, Pascal

A specialized runtime analysis tool on the other hand, will record every memory error with all the details as the error happens. It will put the breakpoints at the exact place where a memory violation happens, or it will allow you to examine the application internals after the run via the recorded runtime analysis data. Runtime analysis removes the guesswork from debugging.

IV. ADVANCED DEBUGGING WITH RUNTIME ANALYSIS

The major goals of debugging are to find the root cause of defects and understand application behavior. Runtime analysis provides additional capabilities that supplement traditional debugging:

- Measurement of runtime parameters, including memory usage, performance, and code coverage.
- Error detection in user code.

A. Code Coverage Analysis

- Finding areas of a program not exercised by a set of test cases,
- Creating additional test cases to increase coverage, and
- Determining a quantitative measure of code coverage, which is an indirect measure of quality.
- Identifying redundant test cases that do not increase coverage. A code coverage analyzer [8] automates this process, e.g.; IBM'S Rational Purify Plus.

B. Memory Leak Description

Memory is allocated but not released causing an application to consume memory reducing the available memory for other applications and eventually causing the system to page virtual memory to the hard drive slowing the application or crashing the application when than the computer memory resource limits are reached. The system may stop working as these limits are approached.

C. Memory Corruption

Memory when altered without an explicit assignment due to the inadvertent and unexpected altering of data held in memory or the altering of a pointer to a specific place in memory.

D. Buffer Overflow

Example 1: Overwrite beyond allocated length overflow.

```
char*a=malloc(128*sizeof(char));  
memcpy(a, data, dataLen); // Error if dataLen too long.
```

Example 2: Index of array out of bounds: (array index overflow - index too large/underflow - negative index)

```
ptr = (char *) malloc(strlen(string_A)); // should be (string_A  
+ 1) to account for null termination.
```

```
strcpy(ptr, string_A); // Copies memory from string_A which  
is one byte longer than it destination ptr [6]. Overflow by one  
byte.
```

E. Using an address before memory is allocated and set

```
struct*ABC_ptr;
x = ABC_ptr->name;
```

In this case the memory location is NULL or random.

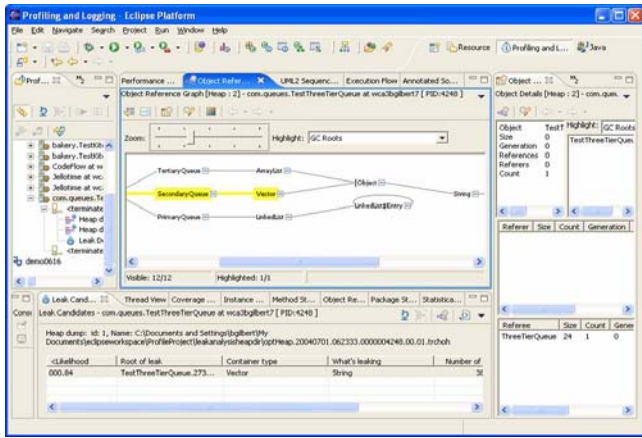


Fig 1: Memory Leak Detection using Rational Application Developer GUI

V. Debugging and testing tools

There are an abundance of software testing tools [7] exist. The correctness testing tools are often specialized to certain systems and have limited ability and generality. Robustness and stress testing tools are more likely to be made generic.

- Run-time analysis tools [5] work at run-time instead of compile time.
- Example -Purify (www.rational.com).
- Purify modifies object files at link time.
- After execution, Purify will report bugs such as memory leaks and null dereferences
- From the purify manual: "Purify checks every memory access operation, pinpointing where errors occur and providing diagnostic information to help you analyze why the errors occur."

Mothora [DeMillo91] is an automated mutation testing tool-set developed at Purdue University. Using Mothora, the tester can create and execute test cases, measure test case adequacy, determine input-output correctness, locate and remove faults or bugs, and control and document the test.

NuMega's Boundschecker [NuMega99] Rational's Purify [Rational99]. They are run-time checking and debugging aids. They can both check and protect against memory leaks and pointer problems.

Ballista COTS Software Robustness Testing Harness [Ballista99]. The Ballista testing harness is a full-scale automated robustness testing tool. The first version supports testing up to 233 POSIX function calls in UNIX operating systems. The second version also supports testing of user functions provided that the data types are recognized by the testing server. The Ballista testing harness gives quantitative measures of robustness comparisons across operating systems. The goal is to automatically test and harden Commercial Off-The-Shelf (COTS) software against robustness failures.

VI. CONCLUSION

Debugging and Runtime analysis expand standard software development activities along one key dimension, concern for quality. It paves the way for achieving higher software quality through better understanding of the internal workings of an application under development. Source code that compiles is not proof of quality; detailed, reliable, and precise runtime performance, memory utilization, and thread and code coverage analysis data are the only way to determine that an application is free of serious errors and will perform efficiently.

REFERENCES

- [1] Laycock, G.T. (1993) (Post script). The theory and practice of specification Based Software Testing . Dept of Computer Science, Sheffield University, UK. <http://www.mcs.le.ac.uk/people/gtl1/thesis.ps.gz>. Retrired 2008-02-13.
- [2] http://www.satisfice.com/articles/requirements_based_testing.pdf. Retrieved 2008-08-19.
- [3] Savenkov, Roman (2008). *How to Become a Software Tester*. Roman avenkov Consulting. p. 168. ISBN 978-0-615-23372-7.
- [4] <http://www.gnu.org/software/gdb/gdb.html>
- [5] <http://www-128.ibm.com/developerworks/edu/i-dw-r-apptuning-i.html>
- [6] <http://www.informit.com/articles/article.asp?p=3064&seqNum=9>
- [7] http://www.eventhelix.com/RealtimeMantra/Basics/debugging_software_crashes_2.htm
- [8] Chilenski1994 John Joseph Chilenski and Steven P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing", *Software Engineering Journal*, September 1994, Vol. 9, No. 5, pp.193-200.
- [9] B. Beizer, *Black Box Testing*. New York: John Wiley & Sons, Inc., 1995.
- [10] A. Bertolino, "Chapter 5: Software Testing," in *IEEE SWEBOK Trial Version 1.00*, May 2001.
- [11] Savenkov, Roman (2008). *How to Become a Software Tester*. Roman avenkov Consulting. p. 168. ISBN 978-0-615-23372-7.

AUTHORS PROFILE

Salim Istaq, Assistant Lecturer, Computer Engineering, Department of Electrical Engineering, Al-Margeb University, Al-Khoms, Libya.
Aufaq Zargar, Assistant Lecturer, Department of Computer Science, Al-Margeb University, Al-Khoms, Libya.