International Journal of Engineering, Science and Mathematics
Vol. 10 Issue 06, June 2021,
ISSN: 2320-0294 Impact Factor: 6.765
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gage as well as in Cabell's Directories of Publishing Opportunities, U.S.A

DEBUGGING TECHNIQUES & ANTI DEBUGGING

AUTHOR-S.SELVAKUMARI

CO-AUTHOR- V. VANEESWARI

R.SATHISHKUMAR

INTRODUCTION-

Interactive debugging uses debugger tools which allow an program's execution to be processed one step at a time and to be paused to inspect or alter its state. Subroutines or function calls may typically be executed at full speed and paused again upon return to their caller, or themselves single stepped, or any mixture of these options. Setpoints may be installed which permit full speed execution of code that is not suspected to be faulty, and then stop at a point that is. Putting a setpoint immediately after the end of a program loop is a convenient way to evaluate repeating code. Watchpoints are commonly available, where execution can proceed until a particular variable changes, and catchpoints which cause the debugger to stop for certain kinds of program events, such as exceptions or the loading of a shared library.

- *debugging* or *tracing* is the act of watching (live or recorded) trace statements, or print statements, that indicate the flow of execution of a process and the data progression. Tracing can be done with specialized tools (like with GDB's trace) or by insertion of trace statements into the source code. The latter is sometimes called *printf debugging*, due to the use of the printf function in C. This kind of debugging was turned on by the command TRON in the original versions of the novice-oriented BASIC programming language. TRON stood for, "Trace On." TRON caused the line numbers of each BASIC command line to print as the program ran.
- *Activity tracing* is like tracing (above), but rather than following program execution one instruction or function at a time, follows program activity based on the overall amount of time spent by the processor/CPU executing particular segments of code. This is typically presented as a fraction of the program's execution time spent processing instructions within defined memory addresses (machine code programs) or certain program modules (high level language or compiled programs). If the program being debugged is shown to be spending an inordinate fraction of its execution time within traced areas, this could indicate misallocation

of processor time caused by faulty program logic, or at least inefficient allocation of processor time that could benefit from optimization efforts.

- *Remote debugging* is the process of debugging a program running on a system different from the debugger. To start remote debugging, a debugger connects to a remote system over a communications link such as a local area network. The debugger can then control the execution of the program on the remote system and retrieve information about its state.
- *Post-mortem debugging* is debugging of the program after it has already crashed. Related techniques often include various tracing techniques like examining log files, outputting a call stack on the crash,[8] and analysis of memory dump (or core dump) of the crashed process. The dump of the process could be obtained automatically by the system (for example, when the process has terminated due to an unhandled exception), or by a programmer-inserted instruction, or manually by the interactive user.
- *"Wolf fence" algorithm:* Edward Gauss described this simple but very useful and now famous algorithm in a 1982 article for Communications of the ACM as follows: "There's one wolf in Alaska; how do you find it? First build a fence down the middle of the state, wait for the wolf to howl, determine which side of the fence it is on. Repeat process on that side only, until you get to the point where you can see the wolf."[9] This is implemented e.g. in the Git version control system as the command *git bisect*, which uses the above algorithm to determine which commit introduced a particular bug.
- *Record and replay debugging* is the technique of creating a program execution recording (e.g. using Mozilla's free rr debugging tool; enabling reversible debugging/execution), which can be replayed and interactively debugged. Useful for remote debugging and debugging intermittent, non-determinstic, and other hard-to-reproduce defects.
- *Time travel debugging* is the process of stepping back in time through source code (e.g. using Undo LiveRecorder) to understand what is happening during execution of a computer program; to allow users to interact with the program; to change the history if desired and to watch how the program responds.
- *Delta Debugging* – a technique of automating test case simplification.[10]:p.123
- *Saff Squeeze* – a technique of isolating failure within the test using progressive inlining of parts of the failing test.[11][12]
- *Causality tracking*: There are techniques to track the cause effect chains in the computation.[13] Those techniques can be tailored for specific bugs, such as null pointer dereferences.[14]
-

Automatic bug fixing[edit]

Automatic bug-fixing is the automatic repair of software bugs without the intervention of a human programmer.[15][16] It is also commonly referred to as *automatic patch generation*, *automatic bug repair*, or *automatic program repair*.[17] The typical goal of such techniques is to automatically generate correct patches to eliminate bugs in software programs without causing software regression.[18]

Debugging for embedded systems

In contrast to the general purpose computer software design environment, a primary characteristic of embedded environments is the sheer number of different platforms available to the developers (CPU architectures, vendors, operating systems, and their variants). Embedded systems are, by definition, not general-purpose designs: they are typically developed for a single task (or small range of tasks), and the platform is chosen specifically to optimize that application. Not only does this fact make life tough for embedded system developers, it also makes debugging and testing of these systems harder as well, since different debugging tools are needed for different platforms.

Despite the challenge of heterogeneity mentioned above, some debuggers have been developed commercially as well as research prototypes. Examples of commercial solutions come from Green Hills Software,[19] Lauterbach GmbH[20] and Microchip's MPLAB-ICD (for in-circuit debugger). Two examples of research prototype tools are Aveksha[21] and Flocklab.[22] They all leverage a functionality available on low-cost embedded processors, an On-Chip Debug Module (OCDM), whose signals are exposed through a standard JTAG interface. They are benchmarked based on how much change to the application is needed and the rate of events that they can keep up with.

In addition to the typical task of identifying bugs in the system, embedded system debugging also seeks to collect information about the operating states of the system that may then be used to analyze the system: to find ways to boost its performance or to optimize other important characteristics (e.g. energy consumption, reliability, real-time response, etc.).

Anti-debugging

Anti-debugging is "the implementation of one or more techniques within computer code that hinders attempts at reverse engineering or debugging a target process".[23] It is actively used by recognized publishers in copy-protection schemas, but is also used by malware to complicate its detection and elimination.[24] Techniques used in anti-debugging include:

- API-based: check for the existence of a debugger using system information
- Exception-based: check to see if exceptions are interfered with
- Process and thread blocks: check whether process and thread blocks have been manipulated
- Modified code: check for code modifications made by a debugger handling software breakpoints
- Hardware- and register-based: check for hardware breakpoints and CPU registers
- Timing and latency: check the time taken for the execution of instructions
- Detecting and penalizing debugger[24]

An early example of anti-debugging existed in early versions of Microsoft Word which, if a debugger was detected, produced a message that said, "The tree of evil bears bitter fruit. Now trashing program disk.", after which it caused the floppy disk drive to emit alarming noises with the intent of scaring the user away from attempting it again.[25][26]

References

1. ^ *"InfoWorld Oct 5, 1981". 5 October 1981. Archived from the original on September 18, 2019. Retrieved July 17, 2019.*
2. ^ *"Archived copy". Archived from the original on 2019-11-21. Retrieved 2019-12-17.*
3. ^ S. Gill, The Diagnosis of Mistakes in Programmes on the EDSAC Archived 2020-03-06 at the Wayback Machine, Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences, Vol. 206, No. 1087 (May 22, 1951), pp. 538-554
4. ^ Robert V. D. Campbell, Evolution of automatic computation Archived 2019-09-18 at the Wayback Machine, Proceedings of the 1952 ACM national meeting (Pittsburgh), p 29-32, 1952.
5. ^ Alex Orden, Solution of systems of linear inequalities on a digital computer, Proceedings of the 1952 ACM national meeting (Pittsburgh), p. 91-95, 1952.
6. ^ Howard B. Demuth, John B. Jackson, Edmund Klein, N. Metropolis, Walter Orvedahl, James H. Richardson, MANIAC doi=10.1145/800259.808982, Proceedings of the 1952 ACM national meeting (Toronto), p. 13-16
7. ^ The Compatible Time-Sharing System Archived 2012-05-27 at the Wayback Machine, M.I.T. Press, 1963
8. ^ *"Postmortem Debugging". Archived from the original on 2019-12-17. Retrieved 2019-12-17.*
9. ^ *E. J. Gauss (1982). "Pracniques: The 'Wolf Fence' Algorithm for Debugging". Communications of the ACM. **25** (11): 780. doi:10.1145/358690.358695. S2CID 672811.*
10. ^ *Zeller, Andreas (2005). Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann. ISBN 1-55860-866-4.*
11. ^ *"Kent Beck, Hit 'em High, Hit 'em Low: Regression Testing and the Saff Squeeze". Archived from the original on 2012-03-11.*
12. ^ *Rainsberger, J.B. (28 March 2022). "The Saff Squeeze". The Code Whisperer. Retrieved 28 March 2022.*
13. ^ *Zeller, Andreas (2002-11-01). "Isolating cause-effect chains from computer programs". ACM SIGSOFT Software Engineering Notes. **27** (6): 1–10. doi:10.1145/605466.605468. ISSN 0163-5948. S2CID 12098165.*
14. ^ *Bond, Michael D.; Nethercote, Nicholas; Kent, Stephen W.; Guyer, Samuel Z.; McKinley, Kathryn S. (2007). "Tracking bad apples". Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07. p. 405. doi:10.1145/1297027.1297057. ISBN 9781595937865. S2CID 2832749.*
15. ^ *Rinard, Martin C. (2008). "Technical perspective Patching program errors". Communications of the ACM. **51** (12): 86. doi:10.1145/1409360.1409381. S2CID 28629846.*

16. ^ *Harman, Mark (2010). "Automated patching techniques". Communications of the ACM.* ***53*** *(5): 108. doi:10.1145/1735223.1735248. S2CID 9729944.*
17. ^ *Gazzola, Luca; Micucci, Daniela; Mariani, Leonardo (2019). "Automatic Software Repair: A Survey" (PDF). IEEE Transactions on Software Engineering.* ***45*** *(1): 34–67. doi:10.1109/TSE.2017.2755013. hdl:10281/184798. S2CID 57764123.*
18. ^ *Tan, Shin Hwei; Roychoudhury, Abhik (2015). "relifix: Automated repair of software regressions". 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE. pp. 471–482. doi:10.1109/ICSE.2015.65. ISBN 978-1-4799-1934-5. S2CID 17125466.*