DOI:10.1145/3186278

Systematic use of proven debugging approaches and tools lets programmers address even apparently intractable bugs.

BY DIOMIDIS SPINELLIS

Modern Debugging: The Art of Finding a Needle in a Haystack

THE COMPUTING PIONEER Maurice Wilkes famously described his 1949 encounter with debugging like this: "As soon as we started programming, [...] we found to our surprise that it wasn't as easy to get programs right as we had thought it would be. [...] Debugging had to be discovered. I can remember the exact instant [...] when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."[37]

Seven decades later, modern computers are approximately one million times faster and also have one million times more memory than Wilkes's Electronic Delay Storage Automatic Calculator, or EDSAC, an early stored-program computer using mercury delay lines. However, in terms of bugs and debugging not much has changed. As developers, we still regularly make mistakes in our programs and spend a large part of our development effort trying to fix them.

Moreover, nowadays, failures can occur nondeterministically in nanosecond time spans within computer systems consisting of thousands of processors spanning the entire planet running software code where size is measured in millions of lines. Failures can also be frighteningly expensive, costing human lives, bringing down entire industries, and destroying valuable property.[22] Thankfully, debugging technology has advanced over the years, allowing software developers to pinpoint and fix faults in ever more complex systems.

One may reasonably wonder how debugging is actually performed in practice. Three recent publications have shed light on a picture full of contrasts. A common theme is that the practice and problems of debugging have not markedly changed over the past 20 years. Michael Perscheid and colleagues at the SAP Innovation Center and the Hasso Plattner Institute in Potsdam, Germany, examined the debugging practices of professional software developers and complemented the results with an online study.[27] They found developers are not trained in debugging, spend 20% to 40% of their work time in it, structure their debugging process following a simplified scientific method (see Figure 1), are proficient in using symbolic debuggers, regularly debug by adding print statements, are unfamiliar with back-

» key insights

- **Targeted software-development process improvements can aid debugging even in cases where their wholesale adoption is impractical.**
- **Debugging benefits from the widespread availability of code, data, and Q&A forums, and programmers can fix many tricky bugs through the generation and analysis of rich datasets.**
- **Modern debugging tools offer powerful and specialized facilities that can save hours of tedious unproductive debugging work.**

IMAGERY BY BOBNEVV AND ART-SONIK

in-time debuggers and automatic fault localization, and consider design, concurrency, and memory faults as the most difficult to debug. The low level of knowledge and use associated with many advanced debugging techniques was also revealed in a mixed-methods study conducted by Moritz Beller and colleagues at the Delft University of Technology, the Netherlands, and myself.[3] In addition, a team led by Marcel Böhme of Monash University, Clayton, VIC, Australia, performed a controlled study by having software professionals fix faults in a carefully constructed benchmark suite of software faults,[6] finding that professionals typically agree on fault locations they identified using trace-based and interactive debugging. However, the study's subjects then went on to implement incorrect fixes, suggesting opportunities for automated regression testing.

Beginners sometimes view debugging as an opaque process of randomly trying things until locating a fault, a method closer to alchemy than to science. Yet debugging can be systematized into the process illustrated by the Unified Modeling Language activity diagram in Figure 1. The first step involves reliably reproducing the failure. It is up to the programmer to produce meaningful results when running experiments to find the failure's cause. Then comes the task of simplifying the failure's configuration into the smallest test case that would still cause the failure to occur.[40] The small test case simplifies and speeds up the programmer's subsequent fault-discovery work. The corresponding steps are outlined in the top part of Figure 1. The next steps, termed the "scientific method of debugging,"[40] are outlined in the bottom part of the figure. In them, the programmer develops a theory about a fault being witnessed, forms a hypothesis regarding the theory's effects, and gathers and tests data against the hypothesis.[24] The programmer repeatedly refines and tests the theory until the cause of the failure is found.

The programmer may sometimes short-circuit this process by guessing directly a minimal test case or the failure's cause. This is fine, especially if the programmer's intuition as an expert provides correct guidance to the cause. However, when the going gets tough, the programmer should humbly fall back on the systematic process instead of randomly poking the software trying to pinpoint the fault through sheer luck.

The goal of this article is to arm software developers with both knowledge-gathering and theory-testing methods, practices, tools, and techniques that give them a fighting chance when struggling to find the fault that caused a failure. Some techniques (such as examining a memory image, still often termed a magnetic memory "core dump") have been with programmers since the dawn of computing. Others (such as reverse debugging) are only now becoming routinely available. And yet others (such as automatic fault localization based on slicing or statistical analysis) do not seem to have caught on.[27] I hope that summarizing here the ones I find through my experience as most effective can improve any programmer's debugging performance.

When the going gets tough, the programmer should humbly fall back on the systematic process instead of randomly poking the software trying to pinpoint the fault through sheer luck.

On the Shoulders of Colleagues

The productivity boost I get as a developer by using the Web is such that I now rarely write code when I lack Internet access. In debugging, the most useful sources of help are Web search, specialized Q&A sites, and source-code repositories. Keep in mind that the terms of a programmer's work contract might prohibit some of these help options.

Web search. Looking for answers on the Web might sound like cheating. But when debugging, the programmer's goal is to solve a problem, not demonstrate academic knowledge and problem-solving skills. If the fastest way to pinpoint and fix a problem is to copy-and-paste an error message in a Web search engine and select the most promising answer, that is what the programmer should do. One can often obtain better results by polishing the query, removing context-dependent data (such as variable or file names) and enclosing the error message in quotes to search for the exact phrase, rather than just the words in it.

Web search typically works when a programmer encounters problems with widely used third-party software. Two possible reasons can yield an unproductive search: First, the programmer may be the first person ever to encounter the problem. This is unlikely with popular software but can happen when working with a cutting-edge release or with a niche or legacy product. There is always an unlucky soul who is first to post about a failure. Second, the error message the programmer is looking for may be a red herring, as with, say, a standard innocuous warning rather than the actual cause of the failure. One must judge search results accordingly.

Q&A sites. The Web can also help a programmer's debugging through Q&A forums (such as a specific product's issue tracker, a company's internal equivalent, or the various https://stackexchange.com/ sites). If the problem is general enough, it is quite likely an expert volunteer will quickly answer the question. Such forums should be used with courtesy and consideration: One should avoid asking an already-answered question, post to the correct forum, employ appropriate tags, ask using a working minimal example, identify a correct answer, and give back to the community by contributing answers to other questions. Writing a good question post sometimes requires significant research.[20] Then again, I often find this process leads me to solve the problem on my own.

Source-code availability. When the fault occurs within open source software, a programmer can use the Web to find, download, and inspect the corresponding code.[31] One should not be intimidated by the code's size or one's personal unfamiliarity. Chances are, the programmer will be looking at only a tiny part of the code around an error message or the location of a crash. The programmer can find the error message by searching through the code for the corresponding string. Crashes typically offer a stack trace that tells the programmer exactly the associated file and line number. The programmer can thus isolate the suspect code and look for clues that will help isolate the flaw. Is some part of the software misconfigured? Are wrong parameters being passed through an API? Is an object in an incorrect state for the method being called? Or is there perhaps an actual fault in the third-party software?

If the bug fix involves modifying open source software, the programmer should consider contributing the fix back to its developers. Apart from being a good citizen, sharing it will prevent the problem from resurfacing when the software is inevitably upgraded to a newer release.

Tuning the Software-Development Process

Some elements of a team's software-development process can be instrumental in preventing and pinpointing bugs. Those I find particularly effective include implementing unit tests, adopting static and dynamic analysis, and setting up continuous integration to tie all these aspects of software development together. Strictly speaking, these techniques aim for bug detection rather than debugging or preventing bugs before they occur, rather than the location of a failure's root cause. However, in many difficult cases (such as nondeterministic failures and memory corruption), a programmer can apply them as an aid for locating a specific bug. Even if an organization's software development process does not follow these guidelines, they can be adopted progressively as the programmer hunts bugs.

Unit tests. It is impossible to build a bug-free system using faulty software components and devilishly difficult to isolate a problem in a huge lump of code. Unit tests, which verify the functionality of (typically small) code elements in isolation, help in both directions,[28] increasing the reliability of routines (functions or methods) they test by guarding their correctness. In addition, when a problem does occur, the programmer can often try to guess what parts may be responsible for it and add unit tests that are likely to uncover it. This way of working gives the programmer a systematic approach for clearing suspect code until hitting the faulty one. The new unit tests the programmer adds also result in a better-tested system, making refactorings and other changes less risky.

When writing unit tests the programmer is forced to write code that is easy to test, modular, and relatively free of side effects. This can further simplify debugging, allowing the programmer to inspect through the debugger how each small unit behaves at runtime, either by adding suitable breakpoints or by directly invoking the code through the debugger's read-eval-print loop, or REPL, facility.

Debugging libraries and settings. Third-party libraries and systems can also aid a fault-finding mission through the debugging facilities they provide. Some runtime libraries and compilers (such as those of C and C++) provide settings that guard against pointer errors, memory buffer overflows, or memory leaks at the expense of lower runtime performance. Compilers typically offer options to build code for debugging by disabling optimizations (aggressive optimizations can confuse programmers when trying to follow the flow of control and data) and by including more information regarding the source code associated with the compiled code. By enabling these settings the programmer is better able to catch many errors.

Static analysis. One can catch some errors before the program begins to execute by reasoning about the program code through what is termed in the software engineering literature "static program analysis." For example, if a method can return a `null` value and this value is subsequently dereferenced without an appropriate check, a static-analysis tool can determine the program could crash due to a `null` pointer dereference. Tools (such as FindBugs[1] and Coverity Scan[5]) perform this feat through multiple approaches (such as heuristics, dataflow or constraint analysis, abstract interpretation, symbolic execution,[8] and type and effect systems).[23] The end result is a list of messages indicating the location of probable faults in a particular program. Depending on the tool, the approach being used, and the program's language, the list may be incomplete—false negative results—or include entries that do not signify errors—false positives. Nevertheless, finding and fixing such errors often prevents serious faults and can sometimes allow the programmer to find a failure's cause.

Dynamic analysis. An alternative approach for analyzing a program's dynamic behavior is to run it under a specialized tool. This is particularly useful when locating a fault involves sophisticated analysis of large and complex data structures that cannot be easily processed with general-purpose command line tools or a small script. Here are some examples of tools a programmer may find useful. In languages compiled with the LLVM Clang front-end the programmer can use AddressSanitizer,[30] while a program runs, to detect many memory-handling errors: out-of-bounds access, use after free, use after scope exit, and double or invalid frees. Another related tool is Valgrind[21] through which one can find potentially unsafe uses of uninitialized values and memory leaks. In addition, Valgrind's Helgrind and data race detector (DRD) tools can help find race conditions and lock order violations in code that uses the POSIX threads API. If the code is using a different thread API, the programmer should consider applying Intel Inspector technology,[29] which also supports Threading Building Blocks, OpenMP, and Windows threads.

Continuous integration. Running static program-analysis tools on code to pinpoint a fault can be like trying to turn the Titanic around after hitting an iceberg. At that point, catastrophic damage has already been done, and it is too late to change the course of events. In the case of a large software codebase, trying to evaluate and fix the scores of error messages spewed by an initial run of a static-analysis tool can be a thorny problem. The developers who wrote the code may be unavailable to judge the validity of the errors, and attempting to fix them might reduce the code's maintainability and introduce even more serious faults. Also, the noise of existing errors hides new ones appearing in fresh code, thus contributing to a software-quality death spiral. To avoid such a problem, the best approach is to integrate execution of static analysis into the software's continuous-integration process,[10] which entails regularly merging (typically several times a day) all developer work into a shared reference version. Running static analysis when each change is made can keep the software codebase squeaky clean from day one.

Making the Software Easier to Debug

Some simple software design and programming practices can make software easier to debug, by providing or configuring debugging functionality, logging and receiving debug data, and using high(er)-level languages. Again, a programmer can selectively adopt these practices during challenging bug-hunting expeditions.

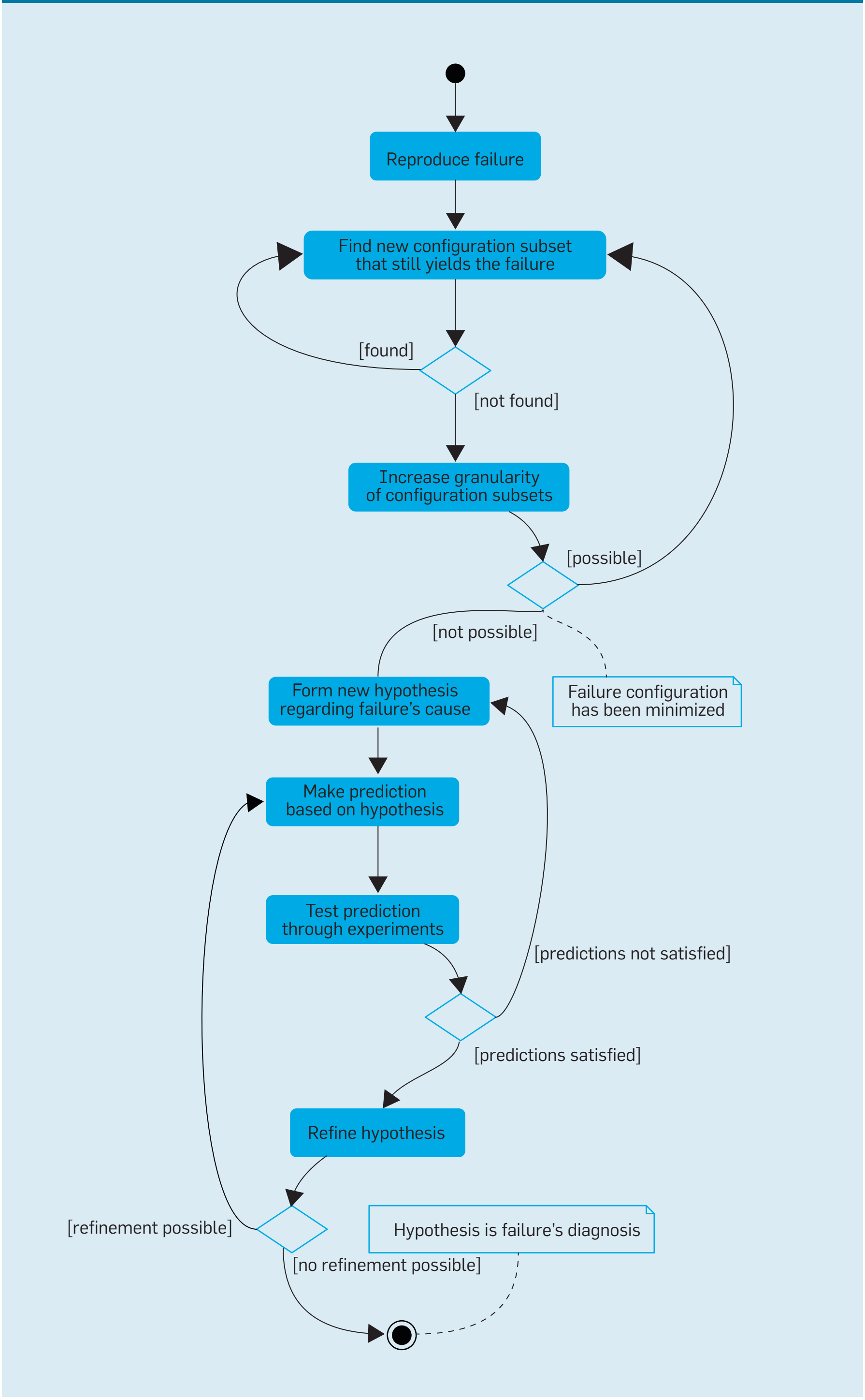
Software's debugging facilities. A helpful way to isolate failures is to build and use debugging facilities within the software. The aim here is to make the software's operation more predictable and transparent. For example, some programs that execute in the background (such as Unix daemons or Windows services) offer a debugging option that causes them to operate synchronously as a typical command-line program, outputting debugging messages on their standard output channel. This is the approach I always use to debug secure shell connection problems. Other debugging settings may make the software's operation more deterministic, which is always helpful when trying to isolate a fault through repeated executions. Such changes may include elimination of multiple threads, use of a fixed seed in random-number generators, and restriction of buffers and caches to a small size in order to increase the likelihood of triggering overflows and cache misses. Software developers should consider adding such facilities to the code they debug. They should, however, keep in mind that some debugging facilities can lead to security vulnerabilities. To avoid this risk, the programmer must ensure the facilities are automatically removed, disabled, or made obnoxiously conspicuous in production builds.

Logging. Programmers are often able to pinpoint faults by adding or using software-logging or -tracing statements.[33] In their simplest form they are plain print commands outputting details regarding the location and state of a program's execution. Unlike watchpoints added in a debugging session, the statements are maintained with the program and are easily tailored to display complex data structures in a readable format.

Modern software tracing is typically performed through a logging framework (such as Apache log4j and Apache log4net) that provides a unified API for capturing, formatting, and handling a program's logging output. Such a framework also allows programmers to tailor at runtime the program's output verbosity and corresponding performance and storage cost. Programmers typically minimize logging to that required for operational purposes when a program executes in a production environment but increase it to include detailed software tracing when they want to debug a failure.

Telemetry. An obvious extension of logging facilities is telemetry, or the

Figure 1. A process for systematic debugging.


Reproduce failure
Find new configuration subset that still yields the failure
[found]
[not found]
Increase granularity of configuration subsets
[possible]
[not possible]
Failure configuration has been minimized
Form new hypothesis regarding failure's cause
Make prediction based on hypothesis
Test prediction through experiments
[predictions not satisfied]
[predictions satisfied]
Refine hypothesis
[refinement possible]
Hypothesis is failure's diagnosis
[no refinement possible]

ability to obtain debugging data from remote program executions (such as those by the program's end users). Ideally, a programmer would want to be able to obtain the following types of data: First, data associated with the execution context (such as the version of the program, helper code, and operating system); values of environment variables; and contents of configuration files. Then comes data about the program's operational status (such as commands executed, settings, and data files). And finally, in cases of program crashes a programmer also needs details regarding the location of the program crash (such as method name, line number, and program counter), runtime context (such as call stack, values of parameters, local variables, and registers), and the reason behind the crash (such as uncaught exception or illegal memory access).

Setting up a telemetry facility before the software is distributed can be a lifesaver when a nasty bug surfaces. On some platforms, a third-party library can be used to collect the required data, outsource its collection, and access the results through a Web dashboard. Keep in mind that telemetry records often contain personal data. When raw memory is recorded, confidential data the reporting code did not gather explicitly (such as passwords and keys) can end up in the telemetry database. Software development teams should consider (carefully) what data to collect, how long to store it, and how they will protect it, and disclose these details to the user.

High-level languages. Another last-resort approach a programmer may have to turn to when debugging complex algorithms, data structures, or protocols may be the implementation of the code in a higher-level formalism. This approach is useful when the programming language developers are working with blurs their focus on the problem's essence. Verbose type declarations, framework boilerplate, unsafe pointers, obtuse data types, or spartan libraries may prevent one from expressing and fixing the parts that matter, burying the programmer instead in a tar-pit of tangential goo. Lifting the code's level of abstraction may simplify finding whether a knotty failure stems from errors in the logic or in the implementation.

My favorite source of intelligence regarding a system's operation is the calls it makes to the operating system.

Suitable technologies for this approach may be scripting languages (such as Python and R), domain-specific languages,[19] or model-driven software development.[35] Adopting high-level formalisms that allow for symbolic reasoning lets the programmer kill two birds with one stone: fix the bug the programmer is after and provide (qualified) guarantees that no other bugs exist in the part of the code the programmer has analytically reasoned about regarding its correctness. Once the alternative implementation is working, the programmer can decide whether architectural, operational, and performance considerations should allow keeping the code in its new formalism, whether to rewrite it (carefully), or automatically transform it to its original programming language.

Insights from Data Analytics

Data is the lifeblood of debugging. The more data that is associated with a failure, the easier it is to find the corresponding fault. Fortunately, nowadays, practically limitless secondary storage, ample main memory, fast processors, and broadband end-to-end network connections make it easy to collect and process large volumes of debugging data. The data can come from the development process (such as from revision-control systems and integrated development environments, or IDEs), as well as from program profiling. The data can be analyzed with specialized tools, an editor, command-line tools, or small scripts.

Some of the processes described here have been systematized and automated by Andreas Zeller of Saarland University, Germany, under the term "delta debugging"[38] and used to locate cause-effect chains in program states[39] and simplify failure-inducing inputs.[41] Although the corresponding tools were mostly research prototypes, the same ideas can be applied on an ad hoc basis to improve the effectiveness of debugging tasks. Consider these representative examples:

Revision-control data. Bugs often occur as the software evolves. By keeping software under version control, using configuration management tools (such as Git, Mercurial, and Subversion), the programmer can dig into a project's history to aid debugging work. Here are some examples: If a program crashes or misbehaves at a particular program line the programmer can analyze the source code to see the last change associated with that line (for example, with the `git blame` command). A review of the change can then reveal that, say, one's colleague who implemented it forgot to handle a specific case. Alternatively, by reading the version-control log of software changes, the programmer can find a recent change that may be related to the failure being witnessed and examine it in detail.

Another neat use of the version-control system for debugging is to automatically find the change that introduced a fault. Under Git the programmer constructs a test case that causes the fault and then specifies it to the `git bisect` command together with a window of software versions where the fault probably appeared. The command will then run a binary search among all the versions within the window in order to determine the exact change that triggered the failure.

Differential debugging. Differences between datasets can also reveal a fault when the programmer can lay hands on a working system and a failing one.[34] The goal is to find where and why the operation of the two systems diverges. The data that can be used for this purpose can come from their generated log files, their execution environment, or traces of their operation. In all cases the programmer must ensure the two system configurations are as similar as possible, apart from exhibiting the failure.

Examining log files for differences can be easily performed by configuring the most detailed logging possible, collecting the logs, removing nonessential differences (or keeping only the pertinent records), and comparing them.

Looking for differences in the environment in which the two systems operate involves examining a program's user input, command-line arguments, environment variables, accessed files (including configuration, executables, and libraries), and associated services.

Investigating differences in the operation of the two systems is more difficult, but thankfully many tools can help. My favorite source of intelligence regarding a system's operation is the calls it makes to the operating system. They (and their results) often determine to a very large extent a program's behavior. Consequently, any divergence between operating system calls is a valuable hint regarding the fault the programmer is trying to locate. For example, the programmer may see that one program tries to open a file that does not exist, times out on a network connection, or runs out of memory. To trace system calls, the programmer can use the strace, ktrace, or truss tools under Unix and similar systems and Procmon[18] under Windows.

The interactions the programmer wants to investigate may also occur at other levels of a system's stack. The programmer can trace calls to dynamically linked libraries with ltrace[7] (Unix) and Procmon (Windows). The programmer can also untangle interactions with services residing on other hosts by examining network packets through Wireshark's[25] nifty GUI. Keep in mind that most relational database systems provide a way to keep and examine a log of executed SQL statements. The programmer can sometimes better understand a program's behavior by obtaining a snapshot of its open files and network connections. Tools that provide this information include lsof (Unix), netstat (Unix and Windows), and tcpvview (Windows). Finally, two tools, DTrace[14] and SystemTap[11] allow the programmer to trace a system's operation across the entire software stack. They should be used if available.

A programmer can also investigate a failing program's trace log without using a working program's log as a reference. However, such an investigation typically requires a deeper understanding of the program's operation, the ability to pinpoint the pertinent log parts, and access to the source code in order to decipher the trace being read. Things to look for in such cases are failing system calls, library calls that return with an error, network timeouts, and empty query result sets.

The log files of the failing system and the working system often differ in subtle ways that hinder their automatic comparison; for example, they may include different timestamps, process identifiers, or host names. The solution is thus to remove the unessential differing fields. The programmer can do that with the editor, Unix filter tools, or a small script. And then apply one of several file-differencing tools to find where the two log files diverge.

Editor tricks. A powerful text editor or IDE can be a great aid when analyzing log data. Syntax coloring can help the programmer identify the relevant parts. With rectangular selections and regular expressions one can eliminate boilerplate or nonessential columns to focus on the essential elements or run a file-difference program on them. The programmer can also identify patterns associated with a bug using search expressions and matched-text highlighting. Finally, by displaying multiple buffers or windows the programmer can visually inspect the details of different runs.

Command-line tools. The programmer can also perform and, more important, automate many of these tasks and much more with Unix-derived command-line filter tools[15,32] available natively or as add-ons on most platforms (such as GNU/Linux, Windows using Cygwin, and macOS). The programmer can easily combine them to perform any imaginable debugging analysis task. This is important, as effective debugging often requires developing and running ad hoc processing tasks.

Here are several examples of how typical Unix command-line tools can be used in a debugging session: The programmer fetches data from the file system using the `find` command from webpages and services using `curl` and from compiled files (depending on the platform) by running `nm`, `javap`, or `dumpbin`. The programmer can then select lines that match a pattern with `grep`, extract fields with `cut`, massage the content of lines with `sed`, and perform sophisticated selections and summarize with `awk`. With normalized datasets at hand, the programmer can then employ `sort` and `uniq` to create ordered sets and count occurrences, `comm` and `join` to find set differences and join sets together, and `diff` to look at differences. Lastly, the number of results can be summarized with `wc`, the first or last records can be obtained with `head` or `tail`, and a hash for further processing derived through `md5sum`. For tasks that are performed often, it is a good practice to package the invocation of the corresponding commands into a Unix shell script and distribute the script as part of the code's software-developer tools.

As a concrete case, consider the task of locating a resource leak in the code. A simple heuristic could involve looking for mismatches between the number of calls to `obtainResource` and calls to `releaseResource`; see Figure 2 for a small Bash script that performs this task. The script uses the `grep` and `sort` commands to create two ordered sets: one with the number of calls to `obtainResource` in each file and another with the corresponding number of calls to `releaseResource`. It then provides the two sets to the `comm` command that will display the cases where records in the two sets do not match.

Scripting languages. If the editor cannot handle the required debugging analysis and the programmer is daunted by the Unix command-line interface, one can also analyze data with a scripting language (such as Python, Ruby, or Perl). Typical tasks that need to be mastered in order to analyze debugging data include sequential reading of text records from a file, splitting text lines on some delimiter, extracting data through regular expression matching, storing data in associative arrays, and iterating through arrays to summarize the results. An advantage of such scripts over other data-analysis approaches is the programmer can easier integrate them within composite project workflows that may involve sending email messages to developers or updating databases and dashboards.

Profiling. When debugging performance issues, the tools the programmer can use various profilers to debug individual processes. The simplest work through sampling, interrupting the program's behavior periodically and giving a rough indication regarding the routines where the program spends most of its time. The programmer thus identifies the routines on which to concentrate optimization efforts. One notch more advanced is graph-based profilers[13] that intercept each routine's entry and exit in order to provide precise details not only of a routine's contribution to the software's overall CPU use but also how the cost is distributed among the routine's callers. An added complication of performance debugging in modern systems is that machine instructions can vary their execution time by at least an order of magnitude based on context. To get to the bottom of such problems the programmer needs to obtain details of low-level hardware interactions (such as cache misses and incorrect jump predictions) through tools that use the CPU's performance counters. These counters tally CPU events associated with performance and expose them to third-party tools (such as the Concurrency Visualizer extension for Visual Studio, Intel's VTune Performance Analyzer, and the Linux `perf` command). For example, performance counters can allow the programmer to detect performance issues associated with false sharing among threads.

Getting More from a Debugger

Given the propensity of software to attract and generate bugs, it is hardly surprising that the capabilities of debuggers are constantly evolving:

Data breakpoints. One impressive facility in many modern debuggers is the ability to break a program's operation when a given value changes, the so-called "data breakpoints." Unlike code or control breakpoints that are easily implemented by patching the code location where a breakpoint is inserted, data breakpoints are difficult to implement efficiently because the corresponding value needs to be checked after each CPU instruction. A debugger could check the value in software by single-stepping through the program's instructions but would reduce its execution speed to intolerable levels.

Many current CPUs instead offer the ability to perform this check through their hardware using so-called write monitors. All the debugger has to do is to set special processor registers with the memory address and the length of the memory area the programmer wants to monitor. The processor will then signal the debugger every time the contents of these locations change. Based on this facility, a debugger can also implement a conditional data breakpoint that interrupts the program's execution when a value satisfies a given condition. The computational overhead in this case is a bit greater because the debugger

Figure 2. Ad hoc location of a probable resource leak.

```
# List non-common lines between the two sets
comm -3 <(
  # Counts per file of obtainResource
  grep -rc obtainResource . | sort) <(
  # Counts per file of releaseResource
  grep -rc releaseResource . | sort)
```

Figure 3. Example of a reverse-debugging session.

```
Breakpoint 1, main () at hairy_code.c:1219
1219            read_data();
(gdb) record
(gdb) next
1220            analyze_data();
(gdb) next
hairy_code: Panic!
1221            display_results();
(gdb) reverse-next
1220            analyze_data();
(gdb) step
analyze_data () at hairy_code.c:1209
1209            if (n == 0)
(gdb) step
1210                    warnx("Panic!");
```

has to check the condition on every change but still orders of magnitude lower than the alternative of checking after every instruction.

Data breakpoints are especially useful when the programmer is unfamiliar with the program's operation and wants to pinpoint what statements change a particular value. They also come in handy in languages that lack memory bounds checking (such as C and C++) in order to identify the cause of memory corruption. All a programmer has to do is set a data breakpoint associated with the corrupted element and wait for the data breakpoint to trigger.

Reverse debugging. Another cool feature available through recent advances in hardware capabilities is "reverse debugging,"[12] or the ability to run code in reverse, in effect traveling back in time. When forward-stepping through the code starting from a statement *A*, a programmer finds statements and variables that can be influenced by *A*, called by researchers a "forward slice."[40] When stepping through code in reverse from *A*, a programmer finds statements and values that could have influenced *A*; this backward slice can help the programmer understand how the program ended up in a specific state.

Reverse debugging is implemented through brute-force computation by having the debugger log the effect of each instruction and thereby obtain the data required to undo it. It has become feasible with fast CPUs and abundant main memory. When debugging a single application, not all actions can be undone; once an operating system call has been performed on a program, effects that cross the debugger's event horizon are there to stay. Nevertheless, the capability can be beneficial when debugging algorithmic code. It is most useful in cases where, while searching for the cause of a failure, a programmer might inadvertently step or glance over the culprit statements. At this point, the programmer can rewind the execution to the point before the culprit statements and move forward again more cautiously.

As an example, consider debugging a problem associated with the display of the cryptic message "Panic!," which appears in hundreds of places within the code. At some point the programmer may be going over code like this

```
read _ data();
analyze _ data();
display _ results();
```

Stepping into each routine to see if it prints the message may take ages. Figure 3 shows the log of a gdb debugger session, demonstrating how a programmer can find the location through reverse debugging; the source code line listed before each `gdb` prompt is the one to be executed next. Graphical interfaces to similar functionality are also available through commercial offerings (such as Microsoft's IntelliTrace for the .NET platform and the Chronon Time Travelling Debugger for the Java ecosystem).

The programmer first sets up gdb for reverse debugging by issuing the `record` command, then runs each function but steps over its innards with the `next` command. Once the programmer sees the "Panic!" message, which is emitted by the `analyze data` routine, the programmer issues the `reverse-next` command to undo the previous `next` and move the execution context again just before the call to the `analyze.data` routine. This time the programmer issues the `step` command to step *into* the routine and find why the message appeared.

Capture and replicate. With multicore processors found in even low-end smartphones today, multithreaded code and the bugs associated with it are a (frustrating) fact of life. Debugging such failures can be difficult because the operation of multithreaded programs is typically nondeterministic; each run of a program executes the threads in a slightly different order that may or may not trigger the bug. I recall the agony of debugging multithreaded rendering code that would occasionally miscalculate just a couple of pixels in a four-million-pixel image. To pinpoint a race condition that exhibits itself in a few nanoseconds within a multi-hour program run, programmers need all the help they can get from powerful software.

Tools helpful in such cases are often those able to capture and replicate in full detail a program's memory-access operations[16] (such as the PinPlay/DrDebug Program Record/Replay Toolkit,[26] which can be used with Eclipse or gdb, and the Cronon recorder and debugger, which works with Java applications). The way programmers work with these tools is to run the application under their control until the failure emerges. This run will generate a recording of the session that can then be replayed under a specialized debugger to locate the statement that causes the failure. With the statement in hand the programmer can look at the program state to see what caused the particular statement to execute or why its execution was not prevented through a suitable lock. These tools thus transform a fleeting nondeterministic failure into a stable one that can be targeted and debugged with ease.

Running and dead processes. Two time-honored but still very useful things programmers can do with a debugger is to debug processes that are already running and processes that have crashed. Debugging a running process is the way to go if it is misbehaving with a failure that is difficult to reproduce. In this case, a programmer would use the operating system's process-display command (such as `ps` under Unix and TaskManager under Windows) to find the numerical identifier of the offending process. The programmer can then fire the debugger, instructing it to debug the process's executable file but also to attach itself to the running process specified by its identifier. From this point onward, programmers can use the debugger as they normally would:

Interrupt stuck program. A programmer can interrupt a stuck program to see at what point the program entered into an endless loop or issued a nonreturning system call;

Add new breakpoints. A programmer can add new breakpoints to see when and how a program reaches a particular code position; and

Examine values. A programmer can examine the values of variables and display the call stack.

Debugging a crashed process allows programmers to perform a post-mortem examination of the facts related to its demise. Some systems allow programmers to launch a debugger at the moment a process crashes. A more flexible alternative involves obtaining an image of the memory associated with the process, the so-called "core dump" (Unix) or Minidump (Windows). This allows the programmer to obtain the dump from a production environment or a customer site and then dissect it on the development environment. There are various methods for obtaining a process's memory dump. On Unix systems, a programmer typically will configure the operating system core file size limit through the system's shell and then wait for the process to crash or send it a `SIGQUIT` signal. On Windows systems a programmer can use the Procdump[18] program to achieve the same results. In both cases, obtaining a memory dump from a still-running but hung process allows the programmer to debug infinite loops and concurrency deadlocks.[9]

Although a memory dump will not allow a programmer to resurrect and step through the execution of the corresponding process, though it is still useful, because the programmer can examine the sequence of calls that were in effect at the point of the crash, the local variables of each routine in that sequence, and the values of global and heap-allocated objects.

No bug can elude a programmer who perseveres.

Debugging Distributed Systems

Modern computing rarely involves an isolated process running on a system that matches a programmer's particular development environment. In many cases, the programmer is dealing with tens to thousands of processes, often distributed around the world and with diverse hardware ranging from resource-constrained Internet of Things (IoT) devices, to smartphones, to experimental and specialized platforms. While these systems are the fuel powering the modern economy, they also present programmers with special challenges. According to the insightful analysis by Ivan Beschastnikh and colleagues at the University of British Columbia, these are heterogeneity, concurrency, distributing state, and partial failures.[4] Moreover, following my own experience, add the likely occurrence of events that would be very rare on an isolated machine, the difficulty of correlating logs across several hosts,[2] and replicating failures in the programmer's development environment.

Remote debugging. The emergence of cloud computing and the IoT have brought with them the necessity of being able to debug systems remotely. A debugger with a graphical interface is not ideal in such situations because it might not be sufficiently responsive when debugging a cloud application across the planet or when a particular IoT platform may lack the power to run it. Consequently, it may make sense for programmers to acquaint themselves with a debugger's command-line interface, as well as the shell commands required to debug more complex systems. An alternative that may sometimes work is a GUI debugger's ability to communicate with a small remote debugger-monitor program the programmer installs and runs at the remote end.

Monitoring. When debugging distributed systems, monitoring and logging are the name of the game. Monitoring will flash a red light when something goes wrong, giving the team an opportunity to examine and understand why the system is misbehaving and thus help pinpoint the underlying cause. In such cases a programmer is often not debugging the code of individual processes but the architecture, configuration, and deployment of systems that may span an entire datacenter or the entire planet. A team can monitor individual failures and performance trends with systems like Nagios, NetData, Ganglia, and Cacti. An interesting approach for generating and thus being able to debug rare failures in complex distributed systems is to cause controlled component failures through specialized software, an approach pioneered by Netflix through its ChaosMonkey.[36]

Event logging. Given that it is not yet possible for a programmer to single-step concurrently through the multitude of processes that might comprise a modern system, when debugging such failures a programmer must rely on event logging, which involves processes logging operational events that target system administrators and reliability engineers. Unlike the software-tracing statements a programmer may use to pinpoint a failure in an individual process, event logging is always enabled in a production environment. By providing "observability," logging can help operations personnel ascertain an application's health status, view its interactions with other processes, and determine changes in a system's configuration. By listing metrics and error messages, logs can reveal a sickly application (such as one with unusually high latency or memory use) or expose one that fails due to insufficient privileges. Such things can help programmers pinpoint a specific application as a contributing factor in a more complex failure.

Virtualization and system simulators. One family of technologies that can help debug software running on hardware that does not match a given development environment includes virtual machines, emulators, and system simulators. With virtual machines and operating system virtualization systems (such as Docker), software development teams can create a single environment that can be used for development, debugging, and production deployment. Such containers are also useful when a programmer wants to find and eliminate configuration-related errors. Moreover, development environments for some commonly used embedded platforms (such as smartphones) come with an emulator, allowing programmers to experience the capabilities of the target hardware from the comfort of a desktop. Finally, when a team is developing software and hardware together, a full system simulator (such as Simics[17]) will provide a high-fidelity view of the complete platform stack.

Conclusion

The number of possible faults in a software system can easily challenge the limits of human ingenuity. Debugging the corresponding failures thus requires an arsenal of tools, techniques, methods, and strategies. Here I have outlined some I find particularly effective, but there are many others I consider useful, as well as many specialized ones that may work wonders in a particular environment.

Each debugging session represents a new venture into the unknown. Programmers should work systematically, starting with an approach that matches the failure's characteristics, but adapt it quickly as they uncover more things about the failure's probable cause. Programmers should not hesitate to switch from Web searching, to logging, to single-stepping, to constructing a unit test, or a specialized tool. No bug can elude a programmer who perseveres. And keep in mind that the joy of fixing a fault is proportional to the work the programmer puts into debugging the failure.

Acknowledgments

My thanks to Moritz Beller, Alexander Lattas, Dimitris Mitropoulos, Tushar Sharma, and the anonymous reviewers for insightful comments on earlier versions of this article. C

References

1. Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., and Pugh, W. Using static analysis to find bugs. *IEEE Software 25*, 5 (Sept. 2008), 22–29.
2. Bailis, P., Alvaro, P., and Gulwani, S. Research for practice: Tracing and debugging distributed systems; programming by examples. *Commun. ACM 60*, 7 (July 2017), 46–49.
3. Beller, M., Spruit, N., Spinellis, D., and Zaidman, A. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden, May 27–June 3). ACM Press, New York, 2018, 572–583.
4. Beschastnikh, I., Wang, P., Brun, Y., and Ernst, M.D. Debugging distributed systems. *Commun. ACM 59*, 8 (Aug. 2016), 32–37.
5. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (Feb. 2010), 66–75.
6. Böhme, M., Soremekun, E.O., Chattopadhyay, S., Ugherughe, E., and Zeller, A. Where is the bug and how is it fixed? An experiment with practitioners. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany, Sept. 4–8). ACM Press, New York, 2017, 117–128.
7. Branco, R.R. Ltrace internals. In *Proceedings of the Linux Symposium*, A.J. Hutton and C.C. Ross, Eds. (Ottawa, ON, Canada, June 27–30, 2007), 41–52; https://www.kernel.org/doc/ols/2007/ols2007v1-pages-41-52.pdf
8. Cadar, C. and Sen, K. Symbolic execution for software testing: Three decades later. *Commun. ACM 56*, 2 (Feb. 2013), 82–90.
9. Cantrill, B. and Bonwick, J. Real-world concurrency. *Commun. ACM 51*, 11 (Nov. 2008), 34–39.
10. Duvall, P.M., Matyas, S., and Glover, A. *Continuous Integration: Improving Software Quality and Reducing Risk.* Pearson Education, Boston, MA, 2007.
11. Eigler, F.C. Problem solving with Systemtap. In *Proceedings of the Linux Symposium*, A. J. Hutton and C. C. Ross, Eds. (Ottawa, ON, Canada, July 19–22, 2006), 261–268; https://www.kernel.org/doc/ols/2006/ols2006v1-pages-261-268.pdf
12. Engblom, J. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference* (Vienna, Austria, Sept. 19–20). Electronic Chips & Systems Design Initiative, Gières, France, 2012, 28–33.
13. Graham, S.L., Kessler, P.B., and McKusick, M.K. An execution profiler for modular programs. *Software: Practice & Experience 13*, 8 (Aug.1983), 671–685.
14. Gregg, B. and Mauro, J. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD.* Prentice Hall Professional, Upper Saddle River, NJ, 2011.
15. Kernighan, B.W. Sometimes the old ways are best. *IEEE Software 25*, 6 (Nov. 2008), 18–19.
16. LeBlanc, T.J. and Mellor-Crummey, J.M. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers C-36*, 4 (Apr. 1987), 471–482.
17. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., and Werner, B. Simics: A full system simulation platform. *Computer 35*, 2 (Feb. 2002), 50–58.
18. Margosis, A. and Russinovich, M.E. *Windows Sysinternals Administrator's Reference.* Microsoft Press, Redmond, WA, 2011.
19. Mernik, M., Heering, J., and Sloane, A.M. When and how to develop domain-specific languages. *ACM Computing Surveys 37*, 4 (Dec. 2005), 316–344.
20. Nasehi, S.M., Sillito, J., Maurer, F., and Burns, C. What makes a good code example?: A study of programming Q&A in StackOverflow. In *Proceedings of the 28th IEEE International Conference on Software Maintenance* (Riva del Garda, Trento, Italy, Sept. 23–30). IEEE Press, 2012, 25–34.
21. Nethercote, N. and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, CA, June 10–13). ACM Press, New York, 2007, 89–100.
22. Neumann, P.G. *Computer Related Risks.* Addison-Wesley, Reading, MA, 1995.
23. Nielson, F., Nielson, H.R., and Hankin, C. *Principles of Program Analysis.* Springer, Berlin, Germany, 2015.
24. O'Dell, D.H. The debugging mind-set. *Commun. ACM 60*, 6 (June 2017), 40–45.
25. Orebaugh, A., Ramirez, G., and Beale, J. *Wireshark & Ethereal Network Protocol Analyzer Toolkit.* Syngress, Cambridge, MA, 2006.
26. Patil, H., Pereira, C., Stallcup, M., Lueck, G., and Cownie, J. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the Eighth Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, ON, Canada, Apr. 24–28). ACM Press, New York, 2010, 2–11.
27. Perscheid, M., Siegmund, B., Taeumel, M., and Hirschfeld, R. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal 25*, 1 (Mar. 2017), 83–110.
28. Runeson, P. A survey of unit-testing practices. *IEEE Software 23*, 4 (July 2006), 22–29.
29. Sack, P., Bliss, B.E., Ma, Z., Petersen, P., and Torrellas, J. Accurate and efficient filtering for the Intel Thread Checker race detector. In *Proceedings of the First Workshop on Architectural and System Support for Improving Software Dependability* (San Jose, CA, Oct. 21–25). ACM Press, New York, 2006, 34–41.
30. Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. Address-Sanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference* (Boston, MA, June 13–15). USENIX Association, Berkeley, CA, 2012, 309–318.
31. Spinellis, D. *Code Reading: The Open Source Perspective.* Addison-Wesley, Boston, MA, 2003.
32. Spinellis, D. Working with Unix tools. *IEEE Software 22*, 6 (Nov./Dec. 2005), 9–11.
33. Spinellis, D. Debuggers and logging frameworks. *IEEE Software 23*, 3 (May/June 2006), 98–99.
34. Spinellis, D. Differential debugging. *IEEE Software 30*, 5 (Sept./Oct. 2013), 19–21.
35. Stahl, T. and Volter, M. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, Inc., New York, 2006.
36. Tseitlin, A. The anti-fragile organization. *Commun. ACM 56*, 8 (Aug. 2013), 40–44.
37. Wilkes, M. *The Birth and Growth of the Digital Computer.* Lecture delivered at the Digital Computer Museum, available through the Computer History Museum, Catalog Number 102695269, Sept. 1979; https://youtu.be/MZGZfsr1KfY
38. Zeller, A. Automated debugging: Are we close? *Computer 34*, 1 (Nov. 2001), 26–31.
39. Zeller, A. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (Charleston, SC, Nov. 18–22). ACM Press, New York, 2002, 1–10.
40. Zeller, A. *Why Programs Fail: A Guide to Systematic Debugging, Second Edition.* Morgan Kaufmann, Burlington, MA, 2009.
41. Zeller, A. and Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering 28*, 2 (Feb. 2002), 183–200.

Diomidis Spinellis (dds@aueb.gr) is a professor in and head of the Department of Management Science and Technology in the Athens University of Economics and Business, Athens, Greece, and author of *Effective Debugging: 66 Specific Ways to Debug Software and Systems*, Addison-Wesley, 2016.