Rogue Wave SOFTWARE

Accelerating Great Code

FINDING HARD-TO-REPRODUCE BUGS WITH REVERSE DEBUGGING

An explanation of the challenges presented by parallel debugging, and the value of a reverse debugging capability

The hardest step in solving software bugs in a parallel programming environment centers on working backward from a software failure to the original program error. Conventional debugging techniques allow users to control program execution only in the forward direction, forcing developers to apply time-consuming methods to attempt to identify the problem.

Reverse debugging technologies have the potential to greatly reduce the time required to identify and solve many of the most difficult bugs by adding the ability to replay parallel program execution.

This white paper explains the challenges presented by parallel debugging and the value of a reverse capability that enables a developer to examine not just the current state of the program, but to follow its logic backward in execution time from the point of failure. This approach achieves significant productivity gains. The paper highlights:

- Challenges of race conditions and other difficult bugs
- The limitations of the classical interactive debugger paradigm
- A new paradigm for debugging based upon recording program execution

DEBUGGING: THE MOST TIME-CONSUMING PHASE OF THE SOFTWARE DEVELOPMENT CYCLE

A survey conducted by Schwartz Communications concluded that debugging is the most time-consuming and costly phase of the software development lifecycle, with 56 percent of respondents citing debugging as the most significant problem they encounter. Only 24 percent said writing original code was the most time-consuming. In fact, 35 percent of respondents said they spend over half their total development time debugging.

This will come as no surprise to Brian Kernighan, co-author of the first book on the C programming language, who famously wrote: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." It is the nature of debugging that correcting the problem is usually trivial while finding the problem is often exceedingly difficult. It is not uncommon for an error in one part of a program to cause failures in a completely different section, thus making it very difficult to track.

Concurrency and multicore implications

The debugging challenge is amplified by the fact that modern computer programs are typically composite systems that rely on many libraries of components. Bugs are often a complex mixture of interactions between library code and program code. The increasing use of concurrency in applications, with multiple threads or processes running simultaneously, compounds this difficulty. In these applications, debugging is complicated by the fact that the behavior of the program may depend on the order and time sequence in which the threads and processes execute on the hardware.

Limitations of conventional debugging software

Conventional software debuggers let you step line-by-line through a program and watch for a bug. For very simple bugs where the crash happens on the same line as or immediately after the error, this approach works well. You merely step through the program to the point where the crash occurs and, with any luck, the error will be obvious.

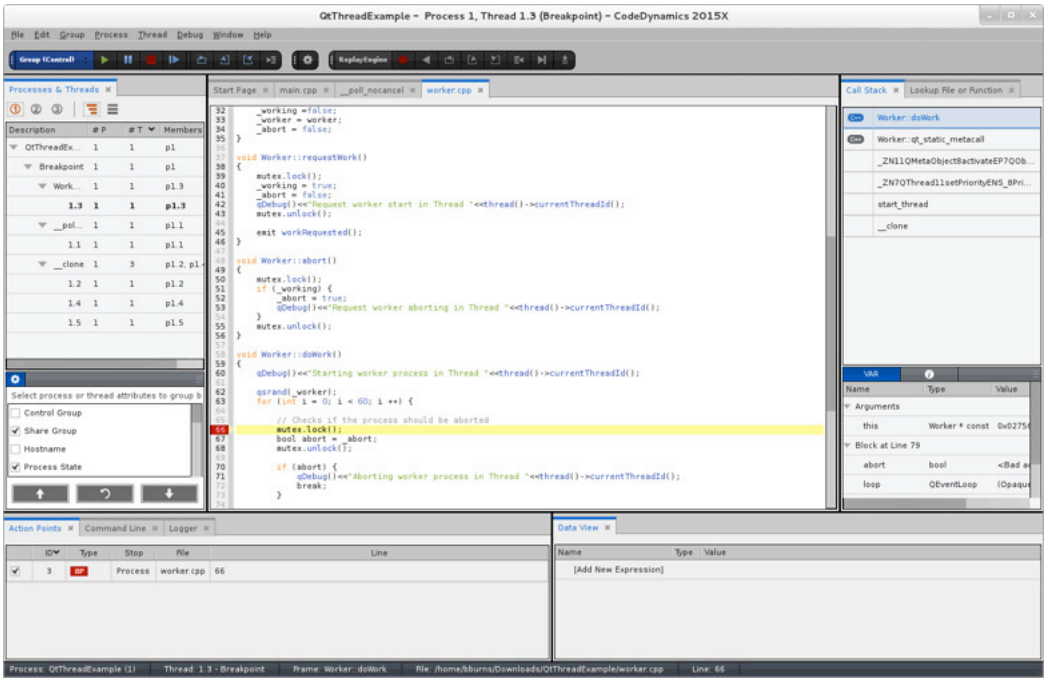
But the conventional debugger is much less effective in the case of other types of bugs, such as those where there is a separation of the error and the point at which execution fails. In this type of bug, a logic error leads to an unexpected state every time input is repeated. The logic error is located far from (in terms of lines of code), but is connected logically to, the failure. You usually address this type of problem with a conventional debugger by running the program to failure and examining the situation to see what the unexpected state is. Normally, the location of the error and your knowledge of the code suggest one or more possible sites for the error. You restart the program in the debugger and run it forward while looking at the logic around these sites to see where it goes off track.

For example, you might examine a failure, see that the crash was caused by a variable being set incorrectly, and restart the program to examine the places where that variable is used, trying to identify the point where it was set incorrectly. Locating that point can be easy or hard, depending on the number of places the variable is accessed, how complex the code is at those sites, and how difficult it is to run the program from site to site without it running away.

This strategy is much harder to apply when the error is separated in both time and program structure from the failure. The error that causes a failure may be in a completely unrelated part of a program. Classic examples of such bugs involve errors in the use of pointers or misuse of the `malloc()` runtime function calls that are used to manage heap memory. Either situation can destroy valid data in a completely unrelated part of the program. In these cases it is very hard to develop any more useful hypothesis than "I bet there is a bad pointer somewhere."

Developers often end up relying on techniques such as looking in a source code management system for the most recently changed parts of the program to decide where to start looking for that pointer error.

Our customers report that although memory leaks and array bounds violation bugs that result from misuse of the `malloc()` system are few in number, they require a majority of their debugging time because they are much harder to resolve with traditional techniques.



Debugging code with multiple threads and processes

The troubleshooting challenge becomes even more daunting when the problem is hard to reproduce. A bug may be hard to pin down because it is very sensitive to complex inputs, or because a race condition exists between components of the program. Race conditions can certainly occur in applications that aren't multithreaded, but they are particularly prevalent when multiple threads or multiple processes make up a program. Problems may occur or not occur on a seemingly random basis based on factors such as which thread wrote to a particular variable first.

When tracking down a bug that involves a race condition you might be tempted to simply run the program over and over again hoping to re-create the condition that causes the error. This can work if the condition happens frequently enough. But, the fact that the debugger is involved can change the timing of the program just enough to alter the probabilities — sometimes radically so. Fundamentally, a race condition has to do with the sequence in which threads execute. A better strategy is often to use a debugger that allows control over thread execution to explore different possible thread sequences. Examination of the code may suggest specific sequences that should be examined first. Discovering a sequence of control operations that cause the error or failure to occur every time goes a long way towards solving the bug.

A NEW PARADIGM, REVERSE DEBUGGING

All of these problems revolve around the nature of conventional debugging in which you must start from the beginning of the program and try to work forward to the cause of the failure.

Instead of going back to the beginning to try and recreate the conditions of a problem, reverse debugging lets you start from the point of failure and work backward in time to find the cause much more easily. Recreating the conditions of the crash, which is sometimes the hardest problem in conventional forward debugging, is no longer necessary.

Reverse debugging helps address the most time-consuming aspects of debugging today's multilevel, multicomponent, multithreaded and multiprocess applications. Since you can move backward, there is no risk that a stochastic problem will disappear. Nor do you have to waste time trying to drive the program over exactly the same path in order to re-create the error. The wide range of options provided by the reverse debugger make it possible to step backward a few lines of code to errors that closely precede the resulting failure or to move intelligently over large swathes of the program to detect errors that occur long before the failure they cause.

What's going on behind the scenes?

Although the idea of reverse debugging sounds simple, actually some very difficult tasks have to be accomplished behind the scenes in order to make it happen. The reverse debugger first records critical aspects of program execution that determine the trajectory of the program. This includes any time that the program reads data or makes a system call that returns data (such as a call to get a pseudo-random number). The information captured during the recording process amounts to a database of known states of the program so that backward-stepping operations or many much larger backward moves can be made with little or no delay.

The reverse debugger provides the ability to instruct the process to go to any previous point in the execution history. Behind the scenes the engine uses the recorded information about the processes and the fact that many operations in a computer are deterministic to run the process to the desired point. At that point, the debugger presents a view of the process as it would normally do, allowing you to examine the stack backtrace, global, heap, and automatic variables, set breakpoints and perform other typical debugging operations.

REVERSE DEBUGGER: REPLAYENGINE

CodeDynamics from Rogue Wave has been extended to provide several new stepping operations with the enablement of a reverse debugger, ReplayEngine. These backward-stepping options can be coupled with the forward-stepping commands to step backward and forward through the program to quickly identify the cause of faults and gain an understanding of the program's behavior.

The simplest approach is to step backward line-by-line through the program in exactly the same way that a conventional debugger lets you step forward through the program. The same rules apply as in forward debugging. When stepping through a loop construct, back stepping from the top line will go first to the conditional, then back into the bottom of the loop.

As with forward stepping, several different backward-stepping options are available. The "back next" operation takes the target program backward over a single executable line of code at the same scope level even if it contains one or more function calls. The "back step" operation takes the target program backward over a line of code unless that line contains one or more functions. In that case it takes the target program to the previously executed line of code even if that line occurs in one of the called subroutines. When executed from the first line of a function, both of these commands will take the program to the point just before the function was called. The "back out" command, available only when working in a function, takes the program to the line just before the current function was called.

ReplayEngine also provides some powerful options for moving backward longer distances in the program. You can select any line in the program prior to the current line and then select the "run back to" operation to move to that point, with the program in the state that it was in when it was most recently at that line. The "run back to" operation works the same way as the "run to" operation except in the opposite direction. This operation is particularly helpful in the situation where the current line is in a loop construct. Rather than stepping around the loop many times to return to an earlier point in the program, you can simply jump to a point before the loop started.

Integration with CodeDynamics

ReplayEngine is provided as an add-on to CodeDynamics. Its capabilities are easily accessed from the graphical user interface of the debugger in a manner that is analogous to existing forward debugging capabilities. Both forward- and backward-stepping operations, as well as other important operations, are accomplished through buttons on a prominent toolbar. During reverse debugging, you have access to the full range of features that power the conventional debugging process.

For example, the windows representing your code and data are all highly interactive. You can use the mouse to drill down on any function, variable, pointer, element, thread, process, or breakpoint in order to see more detail about them. You can easily control and inspect applications that are composed not just of a single process, but sets of thousands of processes running across the many cores of a system. You can work with a single process, work with one process, and any other related processes, or work with all processes that make up the job.

The power of ReplayEngine is increased by the flexibility in representing data sets provided by CodeDynamics. You can call up a graphical representation of the program data (surface plots of arrays), program state (program call graph), communication state (message queues), and memory management state (heap statistics and heap graphical display) to look for incorrect patterns. These capabilities can help flag erroneous patterns by providing statistical analysis of program data, detecting cycles within communication patterns and highlighting leaked blocks of memory.

TIME IS MONEY: IMPROVED PRODUCTIVITY

Reverse debugging technology enables you to find the cause of software bugs much more quickly and directly by moving in a more natural direction from the problem backward to its cause. A conventional debugger puts you in the position of a detective who must analyze circumstantial evidence such as bloodstains and muddy footprints from the scene of a crime. A reverse debugger, on the other hand, provides the equivalent of a security camera that recorded the entire crime, giving you the ability to go backward in time. ReplayEngine can reduce the caseload of unsolved bugs and let you move ahead with your real work — you could say it's "one step backward, two steps ahead."

Rogue Wave provides software development tools for mission-critical applications. Our trusted solutions address the growing complexity of building great software and accelerates the value gained from code across the enterprise. The Rogue Wave portfolio of complementary, cross-platform tools helps developers quickly build applications for strategic software initiatives. With Rogue Wave, customers improve software quality and ensure code integrity, while shortening development cycle times.