BugNet: Recording Application-Level Execution for Deterministic Replay Debugging

WITH SOFTWARE'S INCREASING COMPLEXITY, PROVIDING EFFICIENT HARDWARE SUPPORT FOR SOFTWARE DEBUGGING IS CRITICAL. HARDWARE SUPPORT IS NECESSARY TO OBSERVE AND CAPTURE, WITH LITTLE OR NO OVERHEAD, THE EXACT EXECUTION OF A PROGRAM. PROVIDING THIS ABILITY TO DEVELOPERS WILL ALLOW THEM TO DETERMINISTICALLY REPLAY AND DEBUG AN APPLICATION TO PIN-POINT THE ROOT CAUSE OF A BUG.

Satish Narayanasamy
Gilles Pokam
Brad Calder
University of California, San Diego

Debugging software is challenging because of the increasing complexity of software and hardware, and the diversity of operating systems in use today. These factors make it difficult for software companies to reproduce and fix bugs that occur in released code, especially nondeterministic bugs that occur at a customer site. What makes matters worse is the increase in bugs because of the commercial pressure to release software early, aided and abetted by the ease of delivering software patches over the Internet. Tracking down and fixing these bugs can be a nightmare, costing a significant amount of time and money. These software bugs account for nearly 40 percent of computer system failures[1] and cost the US economy an estimated $59.5 billion annually.[2]

A difficult part of debugging is reproducing the bugs that occur at a customer's site. Most current debugging systems rely on core dumps,[3,4] which contain the final state of the system before the crash. Unfortunately, this solution is woefully inadequate because it is difficult to determine the cause of the error responsible for the core dump, and it provides little help in reproducing the bug. To assist developers, we propose the BugNet architecture to continuously record a program's execution as it runs.[5] The information collected right before a program crash can help deterministically replay the last second of execution preceding the crash. Deterministic replay debugging (DRD) is the ability to replay the exact same sequence of instructions that led up to the crash; it is an effective technique for isolating the bug's source. For the programs we have examined, capturing the last second of execution before the crash was sufficiently long enough to debug the root cause of the crash.

Published by the IEEE Computer Society

0272-1732/06/$20.00 © 2006 IEEE

Using BugNet

BugNet continuously records a program's execution. Its overhead is low enough that it can always be left on without interfering with the execution of the monitored program.

To use BugNet, we start by attaching it to a running application and continuously log the application's execution. BugNet writes the logs to a fixed-sized buffer allocated in main memory. BugNet continuously overwrites this buffer, capturing the last second of execution. If a crash occurs, the logs are dumped to disk. Software developers can then use these logs to deterministically replay the last part of execution that led up to the crash, as many times as required to fix the bug. This ability is vital to debug hard-to-reproduce bugs, such as nondeterministic and multithreaded synchronization bugs.

Focusing only on user code

BugNet focuses on debugging only application-level bugs. Therefore, it supports deterministic replay of only user code and shared libraries, but not the operating system code. However, our approach provides the ability to replay an application's execution across context switches, system calls, interrupts, and direct-memory-access (DMA) transfers. We achieve this without having to track exactly what goes on during these system interactions. In contrast, tracking all the system interactions would require additional hardware support, and the resulting solution would depend heavily on the operating system. Our BugNet solution avoids both of these problems.

Overview of BugNet

BugNet is based on the observation that the following information is sufficient to deterministically replay a program's execution:

- initial architectural state (program counter and register values),
- register and program counter updates from system calls and interrupts, and
- all load values used during the program's execution.

We can easily record the first two items by recording the values in all the registers and the program counter after servicing system calls and interrupts. But recording every load value is too expensive and impractical.

We observe that if we ignore system and multithreading interactions, we need to log the value for a load only if it is the *first access to a memory location*. Values for subsequent loads to the logged memory location can be determined during deterministic replay. To accomplish this, we effectively mark each logged memory address (described in the next section), so that we do not have to log the values loaded from those addresses again. This optimization significantly reduces the amount of information in the log.

The main issue with this optimization is that logging only the first access to a memory location for a user thread will not capture future external updates to that location through operating system or shared-memory interactions. Therefore, we must detect the external updates to an already logged memory location. If the value has changed, we need to log it again. System calls, interrupts, DMA transfers, and shared-memory interactions are the only possible external memory updates. When any of these external effects update a memory location, we unmark that location's address, so that when the user thread accesses that address again, BugNet will log the updated value.

Although this optimization reduces the number of load values that need to be logged, we can further reduce the log size by compressing the load values. We can do so by exploiting value locality across the different load values.

The final component in our architecture relates to logging multithreaded shared-memory dependencies to debug shared-memory interactions and data races. This requires logging the order of memory operations executed across all of the threads in a program's execution. To create this log, we use the technique proposed by Xu et al., which monitors the coherence traffic to capture the shared memory dependencies.[6]

BugNet architecture

We next describe the BugNet logging approach and architecture in more detail.

BugNet checkpoint intervals

BugNet breaks a thread's execution into checkpoint intervals. For the monitored process, BugNet records checkpoint logs sep-

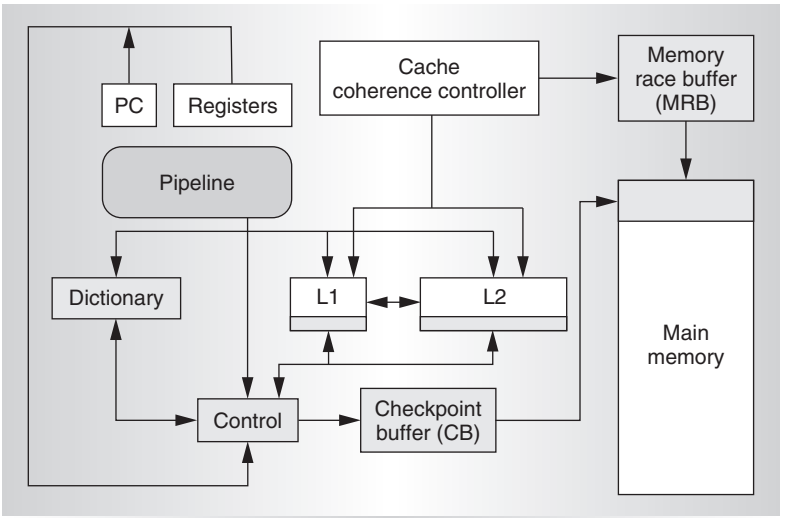

Figure 1. BugNet architecture.

arately for each thread. A *checkpoint interval* represents a window of execution that BugNet captures during logging for each thread. These checkpoints are the instructions a developer can replay during debugging. For a checkpoint interval, BugNet records enough information to start replaying program's execution from the interval's first instruction. This allows us to replay each checkpoint interval independent of the other intervals. The maximum size (in terms of number of committed instructions) for a checkpoint interval is predetermined by the operating system based on the main memory buffer space allocated to record the logs. When a checkpoint interval reaches this limit, BugNet initiates a new checkpoint interval. However, BugNet might prematurely terminate a checkpoint interval on encountering an interrupt, system call, or a context switch (a process that will be described in more detail later). The checkpoint logs are stored in a memory buffer whose size is small enough to avoid degrading the performance of the program being monitored. Our fixed-size memory buffer can maintain only a finite number of checkpoints for each thread.

BugNet components

We built BugNet based on the observation that the load values read by a program drive the program's execution. Therefore, to replay a checkpoint interval, we only need to record the initial register state and then record the values of load instructions executed in that interval.

Figure 1 represents the major components in the BugNet architecture. Components shaded in gray are the new additions to a conventional processor. BugNet creates a checkpoint at the beginning of each checkpoint interval. For every checkpoint, we record a log header that contains the current Thread ID and Process ID, and a Checkpoint ID that BugNet uses to order the checkpoints collected for a thread. The header also contains the initial architectural state, which includes the program counter and the register values. At the start of a checkpoint interval, this header is recorded into the *checkpoint buffer*. After initialization, whenever the processor executes a load instruction, BugNet creates a new log entry to record the load value, which also goes into the checkpoint buffer. The log entry created for a load contains the number of load instructions executed since the last logged load, and a compressed version of the load's value. Since we do not log every load instruction, we must log the number of load instructions to be skipped, during deterministic replay, before consuming the next load log entry. BugNet does not log the addresses for loads because they can be deterministically regenerated during replay.

Recording the resultant value of every load instruction would be too expensive, as described earlier. Instead, we record a load value only if

- the load is the first access to a memory location in the checkpoint interval, or
- the memory value accessed by a load has changed because of an external system effect or a shared-memory thread—since it was last logged in the checkpoint interval.

To approximate the logging of only the first access to each memory location, we associate a bit with every word in the cache. These bits track already logged memory addresses. When a load is executed, if the bit is not set in the cache, BugNet logs the load's value and sets the bit. Instead, if the bit was set, it means that we have already logged this memory location, so BugNet will not log the load's value again. All of these bits are cleared at the start of every checkpoint interval.

To further optimize the log size, we employ

a hardware dictionary compressor, as Figure 1 indicates; it exploits frequently occurring load values. The dictionary keeps track of the most frequent values seen in the checkpoint interval. If a load value appears in the dictionary, we log the index (instead of the full value) into the dictionary.[7]

The *first-load log* (FLL) contains the initial architectural-state information for a checkpoint interval along with all of the load log entries for that interval. Each FLL contains sufficient information to deterministically replay the checkpoint interval. Enough FLLs are kept track of in the memory for the threads being traced so that we can replay millions of instructions executed by each thread. This allows us to reexecute the instructions with exactly the same memory values, and input and output registers as in the original execution. This is true even in the case of multithreaded programs because the FLL for a thread contains all the information necessary to replay each thread independent of the other threads.

Figure 1 shows two first-in first-out queues: the checkpoint buffer and memory race buffer. Both are memory-backed, and are lazily written to memory when the bus is free, but these writes do not go through the cache hierarchy. The checkpoint buffer holds the first load log, and the memory race buffer is used to record the shared-memory dependencies.

The operating system provides support for managing the memory space allocated for BugNet's use. Program execution is continuously logged, overwriting older logs in memory, and logs are dumped to disk only when the execution terminates. Therefore, memory can contain logs corresponding to multiple consecutive checkpoints and logs from many different threads. When the allocated memory space fills up, BugNet discards the log corresponding to the oldest checkpoint for a thread to make space for the newer logs.

System calls, interrupts, and context switches

Our goal is to replay and debug only the application code, so we do not record what goes on during system events. We do not record the value of load instructions executed as part of system calls or interrupts. Nevertheless, we must track how these system interactions affect the application's execution so that we can replay the application across the system events. Interrupts can modify the memory state, and they can even change the architectural state of the program's execution by modifying the program counter or registers.

BugNet solves this problem by prematurely terminating the current checkpoint interval on encountering an interrupt, system call, or context switch.[5] When control returns to the user-level thread, it creates a new checkpoint. By creating a new checkpoint interval, we are guaranteed to have the right program counter value and register values after the interrupt, since they will be logged in the header of the new FLL. Also, the cache bits used to track the already-logged first loads are reset when BugNet creates the new checkpoint. This ensures that BugNet logs the load values necessary to replay the instructions executed after the system call.

Support for multithreaded applications and DMA

We assume a shared-memory multiprocessor system to execute multithreaded programs. In shared-memory multithreaded applications, remote threads executing on other processors can modify shared data during a checkpoint interval. DMA transfers have the same problem. When a remote thread modifies a shared-memory location, it invalidates the corresponding cache block before the update. This resets the bits used to track first loads to that cache block. As a result, future load references to that cache block will result in the FLL recording the remote thread's value. Since DMA uses the same coherency mechanism, DMA transfers will also invalidate the block, resetting the first load bits, so we would correctly see any changes due to DMA transfers.

The FLL corresponding to a checkpoint interval is sufficient to replay the instructions executed in that interval. This is true even in the case of multithreaded applications. We can replay any thread independently of the other threads because the FLL records all the input values required for executing each thread. However, to debug data races, we need the ability to infer the ordering of instructions executed across all the threads. To record shared-memory dependencies across threads, we adapt the Flight Data Recorder (FDR) mechanism,[6] and use a separate memory race log for this information. FDR determines the

shared-memory dependencies for a thread by monitoring its coherence messages and logs those dependencies in the memory race log for each thread. To optimize the memory race log sizes, FDR implemented the Netzer optimization in hardware.[8] We use this method for recording shared-memory dependencies in our memory race logs.[5]

Logging code locations

In addition to data, we must make sure to record information about the code executed during logging. This will allow us to initialize the code space with exactly the same libraries and binaries for deterministic replay debugging.

In BugNet, we assume the support of an operating-system device driver that records information about the loading of all code (static binaries and dynamic libraries) for a monitored program. This information goes into a *code log*. Each entry in the code log contains

- the name and path of the binary or library loaded,
- a checksum to represent the version of the binary or library, and
- the starting address where it was loaded.

When BugNet is enabled for a program's execution, the device driver logs this information for the binary and currently loaded shared libraries. In addition, as the system loads new shared libraries, BugNet will log them. BugNet keeps this code log as long as it is monitoring the program's execution.

Detecting a fault

The operating system terminates threads that perform illegal operations. For example, division by zero or accessing an invalid address can trigger a program crash. Also, the developer can potentially use assertions to terminate the program's execution. Once the operating system detects that the program's execution has terminated abnormally, it records the current instruction count in the checkpoint interval and the address of the faulty instruction in the FLL. We use this information to determine when to stop replaying and to correctly identify the faulty instruction during debugging. After recording the faulty instruction, the operating system collects the first-load logs, and the memory race and code logs for the application from the main memory and hardware buffers. We store the collected logs to disk, and this information is sent to the developer for debugging.

Deterministic replayer

In this section, we describe how to use the BugNet logs to deterministically replay a program's execution. We have implemented a deterministic replayer in Pin[9] that consumes BugNet logs.[5,10]

To replay a checkpoint interval, we start by reading the code log. This restores the code space to its state at the point when the bug occurred. The code log, as described earlier, contains all of the binary and shared-library names, their checksums (versions), and where in the memory they were loaded before the crash occurred. This approach does not support self-modifying code, which we have addressed in our recent work.[11]

After the code space is initialized, the developer chooses the checkpoint interval to start replaying from. We initialize the program counter and the registers with the values found in the header of the checkpoint interval's FLL. Execution starts at the program counter, stepping one instruction at a time. For every load instruction, if the load's value was logged, we obtain the load value from the FLL log and update the simulated memory with this new value. Otherwise, we just use the load value from the simulated memory.

Developers can then step through the program's execution, examining the source lines touched and the variables used. In single stepping through the program's execution, when developers come to the end of a checkpoint, they can just start replaying the next checkpoint. Remember that a checkpoint interval ends after a fixed number of instructions or prematurely terminates because of a system call or an interrupt. From a developer's perspective, the replayer just steps (skips) over the execution of the system call or interrupt. The single stepping can continue until the developer reaches the instruction that caused the crash.

Our logs also support the ability to debug backwards, since we can deterministically recreate the program's execution at any arbitrary point within the collected checkpoint intervals.

Data available during debugging

BugNet logs do not contain a core dump representing the final state of the entire system or main memory. As a result, you cannot examine arbitrary memory locations or data structures during debugging. Instead, developers will only be able to examine the memory values that the program execution touched in the recorded checkpoint intervals. This is what we call the *touched memory image*. If program execution did not access a memory location during the replayed checkpoint intervals, we cannot examine that memory location during replay. This is acceptable, since the memory addresses untouched by the program's execution prior to the crash should not be responsible for the incorrect execution of the program. In fact, having access only to the touched memory image during debugging should help the developers focus only on the variables and data structure fields that caused the bug.

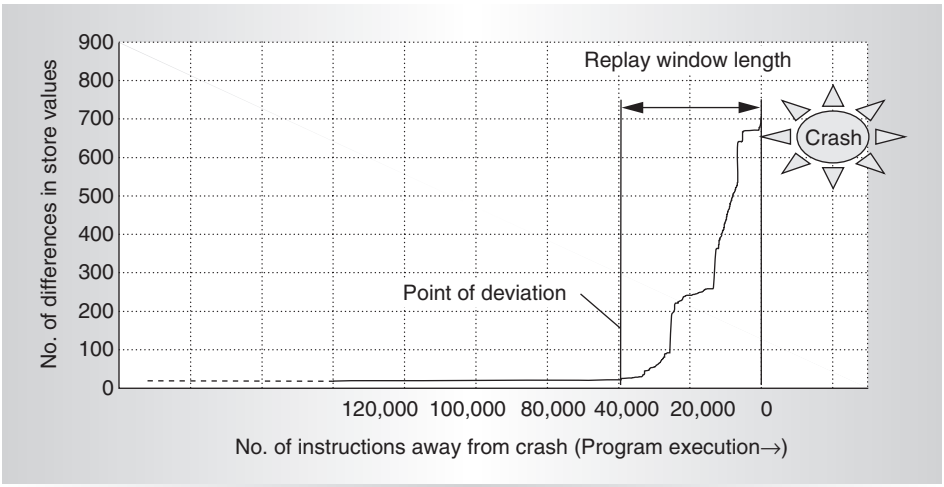

Figure 2. Replay window length. The point of deviation is the point during the buggy program's execution where the store values start deviating from the correct program's execution.

Replay window and log sizes

BugNet is based on the fundamental assumption that the last few moments of a program's execution before a crash are the most critical to fixing a bug. To determine the usefulness of BugNet, we must determine how much to record to fix the majority of bugs.

The *replay window length* is the number of dynamically executed instructions from where the bug manifested itself to where the program crashed. This is the number of instructions that we might need to replay to fix a bug. We originally provided a rough estimate of the lower bound on the replay window length.[5] More recently, we provided an approximation of an upper bound for the replay window length, which we use here.[10]

To analyze the replay window length, we take two binaries corresponding to two versions of the same program. One binary corresponds to the source code that contains the bug, and another corresponds to the same source code but with the bug fixed. We execute the two versions with the same input that exposes the bug. We expect that the two executions will follow similar execution paths up to a point. After that point, the buggy program's execution will start to significantly deviate from the execution of the correct program. We measure how much the buggy program's execution deviates from that of the correct program's execution by comparing the sequences of store values produced by the two executions.[10]

Figure 2 shows an example for a bug in gzip. The *x*-axis represents the buggy program's execution, going from left to right. The *x*-axis indicates the number of dynamic instructions between a point in the program's execution and the crash. The *y*-axis represents the number of differences in the store values between the buggy program's execution and the correct execution. The graph shows the number of mismatches that accumulate over time. Note that the buggy program's output values match those of the correct program's execution for a long time, but it starts to deviate significantly 40,000 instructions before the crash. This is the point of deviation. To fix a bug, a developer should examine the sequence of events right before this point and potentially for some time after this point in the program's execution. Thus, we define the replay window length to be the number of dynamic instructions executed in the buggy program between the point of deviation and the crash or the end of execution.

In Figure 3, we show the replay window

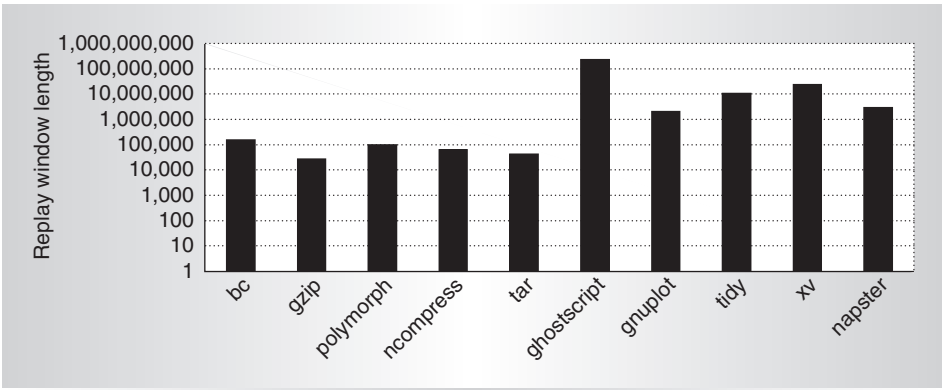

Figure 3. Replay window in dynamic instructions.

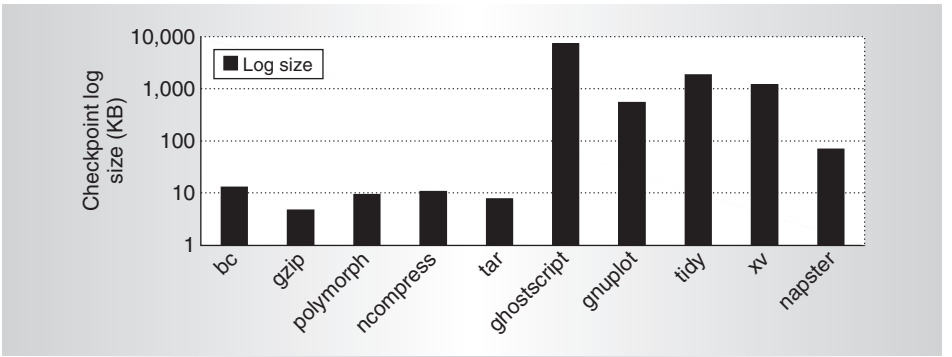

Figure 4. Total size of the FLL checkpoint intervals to capture the full replay window.

lengths required to capture real bugs found in a set of open source programs.[5] All the programs except ghostscript require a replay window of less than 20 million instructions. In contrast, ghostscript (the worst case) requires a replay window of about 230 million instructions. This replay window is an upper bound on the amount of execution a developer would need to fix a bug, and some bugs might be repairable without requiring access to the full replay window.

In Figure 4, we show the FLL size required to capture the replay window in Figure 3. For these results, we used a maximum checkpoint interval size of 10 million instructions. For our worst case, ghostscript, we needed a total of 7.2 Mbytes for the FLL checkpoint intervals to capture the 230 million instruction replay window.

We also examined the logging efficiency for a handful of SPEC programs using BugNet.[5] We found that 0.15 bits per instruction is logged if we use a checkpoint interval of 100 million instructions.

Related work

Several prior techniques look at checkpointing a system's state to replay it later.[6,7,12-15] These systems use a copy-on-write checkpoint scheme similar to SafetyNet.[16] For example, FDR[6] uses the SafetyNet checkpointing scheme to log the values overwritten by the first store to each memory address. With this log and the final system state (core dump), FDR can retrieve a consistent and full memory image prior to the checkpoint's start. In addition, FDR explicitly records external inputs from interrupts, program I/O, and DMA transfers. This allows FDR to support deterministic replay debugging of the entire system, which includes the operating system.

In contrast, BugNet only provides deterministic replay debugging for user code. This provides an easy-to-implement solution for logging the data space that is operating-system independent. Given our model, BugNet cannot replay what goes on in the operating system, but it supports deterministic replay of application code across all forms of system and

shared-memory interactions. Replaying only the application code was sufficient to fix the bugs in all the open source programs that we examined.

Efficient logging of shared-memory dependencies, interrupts, and self-modifying code

We have recently implemented improvements to the BugNet architecture to efficiently log shared-memory dependencies, to provide efficient logging across system calls and interrupts, and to provide support for self-modifying code.[11]

With our BugNet logging approach, we can replay each thread independently. In our recent work,[11] we show how to use the memory traces from replaying each thread independently to come up with a valid ordering of memory operations across a set of shared-memory threads. This allows us to determine cross-thread shared-memory dependencies, without having to record all of the memory dependencies, as in prior work.[5,6]

Our approach of starting a new checkpoint interval for interrupts and system calls is easy to implement, but it could be inefficient if interrupts occur frequently. Each interrupt will restart a checkpoint interval, making it hard to benefit from the first-load optimization. To address this problem, we examined a solution that tracks when we are executing user code as opposed to the system code.[11] When executing system code, every store clears the first load bits in the cache. This guarantees the logging of any memory location updated by system code when it is later loaded by the user code.

A limitation of the BugNet's logging approach[5] is that it does not support self-modifying code. To address this, we support logging the code space[11] in the same way that we log the data space. We treat each instruction fetch just like a load and apply the same first-load logging optimization.

BugNet as a software programmer productivity tool

We have also implemented the BugNet solution completely in software using Pin[9] to provide a deterministic replay debugging tool for software developers.[10] The performance overhead with hardware support for BugNet resulted in only a small slowdown, because the logs are recorded automatically with hardware support and then lazily written back to memory.[5] In comparison, implementing BugNet entirely in software results in an order of magnitude performance slowdown. However, a software-only approach provides programmers and quality assurance teams the means to record a program's execution and then deterministically replay it for debugging. In addition, our software solution provides an additional opportunity to reduce the log sizes and hence has the ability to capture longer replay windows.[10]

We have also examined using BugNet logs to implement operating system independent application-level architecture simulators.[17] One benefit of using the BugNet logging approach is that the simulator does not have to support any infrastructure for emulating the operating system effects. Hence, the BugNet logs provide the ability to simulate real interactive applications across interrupts and DMA transfers.

Support for debugging incorrect output

Thus far, we have only discussed how to deal with bugs that cause a program to crash. But some bugs might lead to erroneous output instead of crashing the program's execution. Both the hardware and software solutions for BugNet are useful for tracking down bugs that lead to incorrect output, but it requires additional support to tell BugNet when to dump the logs. BugNet continuously records recent execution, overwriting logs corresponding to older checkpoint intervals. Therefore, we require system support or a user interface to tell BugNet to dump its recent checkpoint intervals immediately after detecting incorrect program output. We can provide this support by having the user set a break in the program's execution, or by having a developer add software checks or assertions that will be triggered when the program starts producing incorrect results.

Privacy issues

Most customers will not be willing to share their core dumps with developers, since these dumps can contain sensitive information. BugNet does not require core dumps and thereby reduces the amount of information

that customers must pass to a developer. But it still lets the developer see the values of the variables touched in the replay window. Hence, it could be beneficial to obfuscate some of the information in the BugNet logs to protect the privacy of the customer, but still ensure that the developer can replay the program's execution through a path that would expose the bug.

We are entering an era in computer architecture where architects are shifting their focus from pure performance improvements to providing more functionality. Providing hardware support for improved software quality and reliability is becoming a necessity. BugNet is a step in this direction and there are a lot of exciting problems left to be solved. MICRO

Acknowledgments

We thank Jeremy Lau, Mike Stepp, and Cristiano Pereira for providing valuable feedback on this article. This work was funded in part by NSF CCF-0342522, and grants from ST Microelectronics, Intel, and Microsoft.

References

1. E. Marcus and H. Stern, *Blueprints for High Availability,* John Willey & Sons, 2000.
2. G. Tassey, *The Economic Impacts of Inadequate Infrastructure for Software Testing,* Nat'l Institute for Standards Technology, 2002.
3. "Online Crash Analysis," http://oca.microsoft.com/en/dcp20.asp.
4. Netscape Communications Corp., "Netscape Quality Feedback System," http://help.netscape.com/netscape7/qfs3.html.
5. S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 284-295.
6. M. Xu, R. Bodik, and M. Hill, "A Flight Data Recorder for Enabling Full System Multiprocessor Deterministic Replay," *Proc. 30th Ann. Int'l Symp. Computer Architecture* (ISCA 03), ACM Press, 2003, pp. 122-135.
7. G.W. Dunlap et al., "Revirt: Enabling Intrusion Analysis Through Virtual Machine Logging and Replay," *Proc. 5th Symp. Operating System Design and Implementation* (OSDI 02), ACM Press, 2002, pp. 211-224.
8. R.H.B. Netzer, "Optimal Tracing and Replay for Debugging Shared Memory Parallel Programs," *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, ACM Press, 1993, pp. 1-11.
9. C.K Luk et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. ACM/SIGPLAN Conf. on Programming Language Design and Implementation* (SIGPLAN 05), ACM Press, 2005, pp. 190-200.
10. S. Narayanasamy and B. Calder, *Software Profiling for Deterministic Replay Debugging of User Code,* tech. report UCSD-CS2005-0839, Univ. of California, San Diego, 2005.
11. S. Narayanasamy, C. Pereira, and B. Calder, *Efficient Hardware Support for Deterministic Replay Debugging of Shared Memory Dependencies, Interrupts, and Self Modifying Code,* tech. report UCSD-CS2005-0843, Univ. of California, San Diego, 2005.
12. S.M. Srinivasan et al., "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging," *Proc. Usenix 2004 Ann. Technical Conf.*, Usenix Assoc., 2004, pp. 29-44.
13. S. Chen, W.K. Fuchs, and J. Chung, "Reversible Debugging Using Program Instrumentation," *IEEE Trans. Software Eng.*, vol. 27, no. 8, Aug. 2001, pp. 715-727.
14. B. Boothe, "Efficient Algorithms for Bidirectional Debugging, *Proc. ACM/SIGPLAN Conf. on Programming Language Design and Implementation* (SIGPLAN 00), ACM Press, 2000, pp. 299-310.
15. S.T. King, G.W. Dunlap, and P.M. Chen, "Debugging Operating Systems with Time-Traveling Virtual Machines," *Proc. Usenix 2005 Ann. Technical Conf.,* Usenix Assoc., 2005, pp. 1-15.
16. D. J. Sorin et al., "SafetyNet: Improving the Availability of Shared-Memory Multiprocessors with Global Checkpoint/Recovery," *Proc. 29th Ann. Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 123-134.
17. S. Narayanasamy et al., *Automatic Logging of Operating System and Multithreading Effects to Guide Application-Level Architecture Simulation,* tech. report CS2005-0840, Univ. of California, San Diego, 2005.

Satish Narayanasamy is a PhD candidate in the computer science department at the University of California, San Diego. His research interests include computer architecture, hardware and software support to improve programmer productivity, and system security. Narayanasamy has an MS from the University of California, San Diego, and a BE from Anna University, the College of Engineering, Guindy, Chennai, India, both in computer science. He is a member of ACM and IEEE.

Gilles Pokam is a postdoctoral researcher in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include computer architecture and parallel programming models. Pokam has an MS in computer engineering from the Technical University of Berlin, Germany, and a PhD in computer science from the University of Rennes I, France. He is a member of IEEE and ACM.

Brad Calder is a professor of computer science and engineering at the University of California, San Diego. His research interests include computer architecture, compilers, and systems. Calder has a PhD in computer science from the University of Colorado, Boulder, and a BS in computer science and a BS in mathematics from the University of Washington. He is a recipient of an NSF Career Award and is a senior member of IEEE and member of ACM.

Direct questions and comments about this article to Brad Calder, University of California, San Diego, Department of Computer Science and Engineering, 9500 Gilman Dr., La Jolla, CA 92093-0404; calder@cs.ucsd.edu.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.

THE IEEE'S 1ST ONLINE-ONLY MAGAZINE

IEEE
distributed systems
Expert-authored articles and resources ONLINE

IEEE Distributed Systems Online brings you peer-reviewed articles, detailed tutorials, expert-managed topic areas, and diverse departments covering the latest news and developments in this fast-growing field.

Log on for *free access* to such topic areas as

Grid Computing • Middleware Cluster Computing • Security • Peer-to-Peer and More!

To receive monthly updates,email

dsonline@computer.org

http://dsonline.computer.org