Empowering Software Debugging Through Architectural Support for Program Rollback

Radu Teodorescu and Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Abstract

This paper proposes the use of processor support for program rollback, as a key primitive to enhance software debugging in production-run environments. We discuss how hardware support for program rollback can be used to characterize bugs on-the-fly, leverage code versioning for performance or reliability, sandbox device drivers, collect monitoring information with very low overhead, support failure-oblivious computing, and perform fault injection. To test our ideas, we built an FPGA prototype. We run several buggy applications on top of a version of Linux.

1. Introduction

Dynamic bug-detection tools (e.g., [9, 13]) face major challenges when targeting *production-run environments*. In such environments, bug monitoring and detection have to be done with very low overhead. In addition, it is often desirable to provide graceful recovery from bugs, so that the system can continue to work.

One way to accomplish these goals is to provide hardware support in the processor for low-overhead software bug characterization, and for graceful recovery from bugs. For this, we propose a hardware primitive that quickly undoes (rolls back) sections of code. When a certain suspicious event that may be a bug has been detected, the hardware rolls the program thousands of instructions back with very little overhead. At that point, several options are possible. We can either choose to re-execute the same section of code or to jump off to another section where additional monitoring can be done. If we choose the former, we can re-execute the code section with the same input data set but with more instrumentation enabled, so that we can further characterize the bug. Alternatively, we can re-execute the section with a different input or algorithm, to skip the bug altogether.

To test these ideas, we have implemented such a hardware extension to a simple processor prototyped using FPGAs (Field Programmable Gate Arrays). In this paper, we describe the operation and software interface of our prototype. In addition,, we describe some of the uses that such hardware support can have in helping software debugging. Such uses are to fully characterize a bug on-the-fly, leverage code versioning, sandbox the kernel's device drivers, collect and sample information with very low overhead, support failure-oblivious computing, and perform fault injection, among other issues.

This paper also evaluates the FPGA-based prototype we built [16]. The extensions added include holding speculative data in the cache, register checkpointing, and software-controlled transitions between speculative and non-speculative execution. We experiment with several buggy applications running on top of a version of Linux. Overall, we show that this rollback primitive can be very effective in production-run environments.

2 System overview

The system we propose allows the rollback and re-execution of large sections of code (typically up to tens of thousands of instructions) with very low overhead. This is achieved through a few relatively simple changes to an existing processor.

We have implemented two main extensions: (1) the cache can hold speculative data and, on demand, quickly commit it or discard it all, and (2) the register state can be quickly checkpointed into a special storage and restored from there on demand. These two operations are done in hardware. When entering speculative execution, the hardware checkpoints the registers and the cache starts buffering speculatively written data. During speculative execution, speculative data in the cache gets marked as such and is not allowed to be displaced from the cache. When transitioning back to normal execution, any mark of speculative

data is deleted and the register checkpoint is discarded. If a rollback is necessary, the speculatively written data is invalidated and the register state is restored from the checkpoint.

2.1 Speculative execution control

The speculative execution can be controlled either in hardware or in software. There are benefits on both sides and deciding which is best is dependent on what speculation is used for.

2.1.1 Hardware control

If we want the system to always execute code speculatively and be able to guarantee a minimum rollback window, the hardware control is more appropriate. As the program runs, the cache buffers the data generated and always marks them as speculative. There are always two *epochs* of speculative data buffered in the cache at a time, each one with a corresponding register checkpoint. When the cache is about to get full, the earliest epoch is committed, and a new checkpoint is created. With this support, the program can always roll back at least one of the two execution epochs (a number of instructions that filled roughly half of the L1 data cache).

2.1.2 Software control

If, on the other hand, we need to execute speculatively only some sections of code, and the compiler or user is able to identify these sections, it is best to expose the speculation control to the software. This approach has two main benefits: more flexibility is given to the compiler and a smaller overhead is incurred since only parts of the code execute speculatively.

In this approach, the software explicitly marks the beginning and the end of the speculative section with *BEGIN_SPEC* and *END_SPEC* instructions. When a *BEGIN_SPEC* instruction executes, the hardware checkpoints the register state and the cache starts buffering data written to the cache, marking them as speculative.

If, while executing speculatively, a suspicious event that may be a bug is detected, the software can set a special *Rollback* register. Later, when *END_SPEC* is encountered, two cases are possible. If the Rollback register is clear, the cache commits the speculative data, and the hardware returns to the normal mode of execution. If, instead, the Rollback register is set, the program execution is rolled back to the checkpoint, and the code is re-executed, possibly with more instrumentation or different parameters.

If the cache runs out of space before the *END_SPEC* instruction is encountered, or the processor attempts to perform an uncacheable operation (such as an I/O access), the processor triggers an exception. The exception handler decides what to do, one possibility being to commit the current speculative data and continue executing normally.

3 Using program rollback for software debugging

The architectural support presented in this work provides a flexible environment for software debugging and system reliability. In this section, we list some its possible uses.

3.1 An integrated debugging system

The system described here is part of a larger debugging effort for production-run codes that includes architectural and compiler support for bug detection and characterization. In this system, program sections execute in one of three states: *normal*, *speculative* or *re-execute*. While running in speculative mode, the hardware guarantees that the code (typically up to about tens of thousands of instructions) can be rolled back with very low overhead and re-executed. This is used for thorough characterization of code sections that are suspected to be buggy.

The compiler [6] is responsible for selecting which sections of code are more error-prone and thus, should be executed in speculative mode. Potential candidates are functions that deal with user input, code with heavy pointer arithmetic, or newer, less tested functions. The programmer can also assist by indicating the functions that he or she considers less reliable.

In addition, a mechanism is needed to detect potential problems, and can be used as a starting point in the bug detection process. Such a mechanism can take many forms, from a simple crash detection system to more sophisticated anomaly detection mechanisms. Examples of the latter include software approaches like Artemis [3] or hardware approaches like iWatcher [19].

Artemis is a lightweight run-time monitoring system that uses execution contexts (values of variables and function parameters) to detect anomalous behavior. It uses training data to learn the normal behavior of an application and will detect unusual situations. These situations can act as a trigger for program rollbacks and re-executions for code characterization.

iWatcher is an architecture proposed for dynamically monitoring memory locations. The main idea of iWatcher is to associate programmer-specified monitoring functions with monitored memory objects. When a monitored object is accessed, the monitoring function associated with this object is automatically triggered and executed by the hardware without generating an exception to the operating system. The monitoring function can be used to detect a wide range of memory bugs that are otherwise difficult to catch.

We provide two functions, namely `enter_spec` to begin speculative execution and `exit_spec` to end it with commit or rollback. In addition, we have a function `proc_state()` used to probe the state of the processor. A return value of 0 means normal mode, 1 means speculative mode, and 2 means re-execute mode (which follows a rollback).

The following code shows how these functions are used. `exit_spec` takes one argument, `flag`, that controls whether speculation ends with commit or rollback. If an anomaly is detected, the software immediately sets the `flag` variable. When the execution finally reaches `exit_spec`, a rollback is triggered. The execution resumes from the `enter_spec` point.

```
num=1;
...
/* begin speculation */
enter_spec();
...
/* heavy pointer arithmetic */
p=m[a[*x]]+&y;
if (err) flag=1;
...
/* info collection */
/* only in re-execute mode */
if (proc_state()==REEXECUTE) {
 collect_info();
}
exit_spec(flag);
/* end speculation */
num++;
...
```

The compiler inserts code in the speculative section to collect relevant information about the program execution that can help characterize a potential bug. This code is only executed if the processor is in re-execute mode (proc_state() returns 2).

Figure 1 shows the three possible execution scenarios for the example given above. Case (a) represents normal execution: no error is found, the `flag` variable remains clear and, when `exit_spec(flag)` is reached, speculation ends with commit.

In case (b), an abnormal behavior that can lead to a bug is encountered. `Flag` is set when the anomaly is detected and, later, when execution reaches `exit_spec(flag)`, the program rolls back to the beginning of the speculative region and continues in re-execute mode. This can be repeated, possibly even inside a debugger, until the bug is fully characterized. `Flag` can be set as a result of a failed assertion or data integrity test.

Finally, in case (c) the speculative state can no longer fit in the cache. The overflow is detected by the cache controller and an exception is raised. The software is expected to handle this case. The example assumes that the exception handler commits the speculative data. When the execution reaches the `exit_spec(flag)` instruction, the state

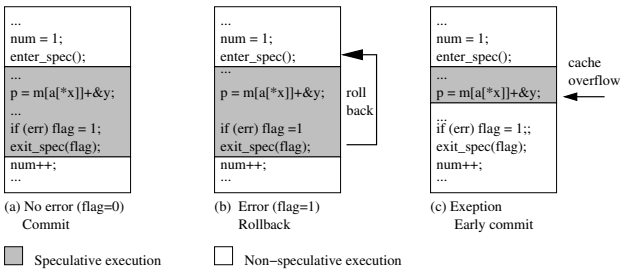

Figure 1. Speculative execution ends with commit (a), a rollback (b), or an early commit due to cache overflow (c).

of the processor is first checked. Since the processor is no longer speculative (due to the early commit), the instruction is simply ignored.

3.2 Other uses of program rollback

3.2.1 Code versioning

Code versioning, or N-version programming [8] is a technique that involves generating multiple, different versions of the same code. It can be used for performance or reliability. When targeting performance, a compiler generates a main version that is aggressively optimized, and potentially sometimes incorrect. Using our hardware, this version can be executed speculatively, with some verification code in place. If the function fails or produces an incorrect result as indicated by the verification code, the processor is rolled back, and a second, unoptimized but safe version of the code is executed.

In the same way, when targeting reliability, we can have two versions of the same function that are safe, have similar performance, but use different functional units in the processor. Each version includes some verification code that checks that the computation was correct. We can first run the first function and its verification code. If the verification code fails, we then run the second function and its verification code. Since the functions use different parts of the processor, they are unlikely to both fail.

3.2.2 OS kernel and driver debugging

One of the major challenges in OS reliability is to ensure correct execution of the OS kernel in the presence of faulty drivers. In fact, in Linux, the frequency of coding errors is seven times higher for device drivers than for the rest of the kernel [1]. Several solutions have been proposed to this problem including many that involve isolating the kernel from the device drivers with some protection layer [15].

In general, these solutions require major changes to OS design and implementation and can introduce significant overheads.

We propose a simpler solution with potentially very low overhead that takes advantage of the rollback support implemented in the hardware.

In general, the kernel and driver code interact through interface functions, and maintain data structures in both kernel and driver memory. In a system like ours, function calls from kernel to driver or vice-versa could be executed speculatively. If an error is detected, the changes made to kernel memory would then be rolled back. The idea is to prevent the kernel from becoming corrupted or even crashing due to a faulty driver. A cleanup procedure could then be called to shut down the driver and either attempt to reinitialize it or report the error to the user.

The current system cannot roll back any I/O operations. This is because we currently buffer only cacheable data. However, we can still roll back the processor in case of a fault. Any communication with the faulty device is lost but the processor is restored to the state before the device access began. If the device somehow corrupted the kernel, the correct state can still be recovered from the checkpoint. The fault model for a system like this would target kernel integrity rather than guaranteeing the correct operation of individual devices.

3.2.3 Lightweight information collection and sampling

Detecting bugs in production code can be challenging because it is hard to obtain substantial information about program execution. It is hard to collect relevant information without incurring a large overhead. Previous solutions to this problem have suggested using statistical sampling to obtain execution information with small overheads [7].

We propose using our system to perform lightweight collection of execution information based on anomaly detection. In this case, the processor would always execute in speculative state. When an anomaly is detected (an unusual return value, a rarely executed path, etc.), the processor is rolled back as far as its speculative window allows and then re-executed. Upon re-execution, instrumentation present in the code is turned on, and the path that led to the anomalous execution recorded. This allows more precise information about anomalous program behavior than statistical sampling would. Also, because the additional code is rarely executed, the overhead should be very low.

3.2.4 Failure-oblivious computing

A failure-oblivious system [12] enables programs to continue executing through memory errors. Invalid memory accesses are detected, but, instead of terminating the execution or raising an exception, the program discards the invalid writes and manufactures values for invalid reads, enabling the program to continue execution.

A failure-oblivious system can greatly benefit from our rollback support. When a read results in an invalid access, the system enters speculative mode, generates a fake value, and uses it in order to continue execution. It is unknown however, whether the new value can be used successfully or, instead, will cause further errors. Since the code that uses the fake value executes speculatively, it can roll back if a new error is detected. Then, the program can use a different predicted value and re-execute the code again, or finally raise an exception.

3.2.5 Fault injection

Our rollback hardware can also be used as a platform for performing fault injection in *production systems*. It offers a way of testing the resilience of systems to faulty code, or test *what if* conditions, without causing system crashes. The code that is injected with faults is executed speculatively, to determine what effect it has on the overall system. Even if the fault propagates, the code can be rolled back and the system not allowed to crash. The process can be repeated multiple times, with low overhead, to determine how a system behaves in the presence of a wide array of faults.

4 Evaluation

4.1 FPGA infrastructure

As a platform for our experiments, we used a synthesizable VHDL implementation of a 32-bit processor [4] compliant with the SPARC V8 architecture.

The processor has an in-order, single-issue, five stage pipeline. This system is part of a system-on-a-chip infrastructure that includes a synthesizable SDRAM controller, PCI and Ethernet interfaces. The system was synthesized using Xilinx ISE v6.1.03. The target FPGA chip is a Xilinx Virtex II XC2V3000 running on a GR-PCI-XC2V development board [10].

On top of the hardware, we run a version of the SnapGear Embedded Linux distribution [2]. SnapGear Linux is a full source package, containing kernel, libraries and application code for rapid development of embedded Linux systems. A cross-compilation tool-chain for the SPARC architecture is used for the compilation of the kernel and Linux applications.

To get a sense of the hardware overhead imposed by our scheme, we synthesize the processor core with and without the support for speculative execution. We look at the utilization of the main resources in FPGA chips, the Configurable Logic Blocks (CLBs). Virtex II CLBs are organized in an array and are used to build the combinatorial

and synchronous logic components of the design. The CLB overhead of our scheme is small (less than 4.5% on average) [16].

4.2 Speculative execution of buggy applications

We run experiments, using standard Linux applications that have known (reported) bugs. For these applications, we want to determine whether we can speculatively execute a section of dynamic instructions that is large enough to contain *both the bug and the location where the bug is caught* by a detection mechanism like iWatcher [19]. Some parameters of the experimental setup are given in Table 1.

We assume that the compiler has identified the suspicious region of code that should be executed speculatively. We also assume the existence of a detection mechanism (such as iWatcher), which can tell us that a bug has occurred. We want to determine if, under these circumstances, we can roll back the buggy section of code in order to characterize the bug thoroughly by enabling additional instrumentation.

We use five buggy programs from the open-source community. The bugs were introduced by the original programmers. They represent a broad spectrum of memory-related bugs. The programs are: *gzip, man, polymorph, ncompress and tar*. *Gzip* is the popular compression utility, *man* is a utility used to format and display on-line manual pages, *polymorph* is a tool used to convert Windows style file names to something more portable for UNIX systems, *ncompress* is a compression and decompression utility, and *tar* is a tool to create and manipulate archives.

In the tests we use the bug-exhibiting inputs to generate the abnormal runs. All the experiments are done under realistic conditions, with the applications running on top of a version of Linux running on our hardware.

Table 1. Main parameters of the experimental setup.

Processor	LEON2, SPARC V8 compliant
Clock frequency	40MHz
Instruction cache	8KB
Data cache	32KB
RAM	64MB
Windowed register file	8 windows $\times$ 24 registers each
Global registers	8 registers

Table 2 shows that the buggy sections were successfully rolled back in most cases, as shown in column four. This means that the system executed speculatively the entire buggy section, performed a rollback when the end speculation instruction was reached, and then re-executed the entire section. On the other hand, a failed rollback (*polymorph*) means that before reaching the end speculation instruction, a condition is encountered that forces the early commit of the speculative section. Rollback is no longer possible in this case.

The fifth column shows the number of dynamic instructions that were executed speculatively. Notice that in the case of *polymorph* the large number of dynamic instructions causes the cache to overflow the speculative data, and forces an early commit.

5 Related work

Some of the hardware presented in this work builds on extensive work on Thread-Level Speculation (TLS) (e.g. [5, 14]). We employ some of the techniques first proposed for TLS to provide lightweight rollback and replay capabilities. TLS hardware has also been proposed as a mechanism to detect data races on-the-fly [11].

Previous work has also focused on various methods for collecting information about bugs. The "Flight Data Recorder" [17] enables off-line deterministic replay of applications and can be used for postmortem analysis of a bug. It has a significant overhead that could prevent its use in production codes.

There is other extensive work in the field of dynamic execution monitoring. Well-known examples include tools like Eraser [13] or Valgrind [9]. Eraser targets detection of data races in multi-threaded programs. Valgrind is a dynamic checker to detect general memory-related bugs such as memory leaks, memory corruption and buffer overflow. Most of these systems have overheads that are too large to make them acceptable in production code.

There have also been proposals for hardware support for detecting bugs, such as iWatcher [19] and AccMon [18]. These systems offer dynamic monitoring and bug detection capabilities that are sufficiently lightweight to allow their use on production software. This work is mostly complementary to ours. In fact we assume some of the detection capabilities of iWatcher when evaluating our system.

6 Conclusions and future work

This work shows that with relatively simple hardware we can provide powerful support for debugging production codes. We show it by building a hardware prototype of the envisioned system, using FPGA technology. Finally, we run experiments on top of Linux running on this system.

The hardware presented in this work is part of a comprehensive debugging infrastructure. We are working toward integrating compiler support to identify vulnerable code regions as well as to instrument the code with speculation control instructions.

Application	Bug location	Bug description	Successful rollback	Speculative instructions
ncompress-4.2.4	compress42.c: line 886	Input file name longer than 1024 bytes corrupts stack return address	Yes	10653
polymorph-0.4.0	polymorph.c: lines 193 and 200	Input file name longer than 2048 bytes corrupts stack return address	No	103838
tar-1.13.25	prepargs.c: line 92	Unexpected loop bounds causes heap object overflow	Yes	193
man-1.5h1	man.c: line 998	Wrong bounds checking causes static object corruption	Yes	54217
gzip-1.2.4	gzip.c: line 1009	Input file name longer than 1024 bytes overflows a global variable	Yes	17535

Table 2. Speculative execution in the presence of bugs.

We have presented several uses of this hardware for debugging, including to characterize bugs on-the-fly, leverage code versioning for performance or reliability, sandbox device drivers, collect monitoring information with very low overhead, support failure-oblivious computing, and perform fault injection. We will be implementing some of these techniques in the near future.

References

[1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. *SIGOPS Operating Systems Review*, 35(5):73–88, 2001.

[2] CyberGuard. SnapGear Embedded Linux Distribution. www.snapgear.org.

[3] L. Fei and S. P. Midkiff. Artemis: Practical Runtime Monitoring of Applications for Errors. Technical Report TR-ECE05-02, School of ECE, Purdue University, 2005.

[4] J. Gaisler. LEON2 Processor. www.gaisler.com.

[5] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, pages 866–880, September 1999.

[6] S. I. Lee, T. A. Johnson, and R. Eigenmann. Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *Languages and Compilers for Parallel Computing (LCPC)*, pages 539–553, 2003.

[7] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Not.*, 38(5):141–154, 2003.

[8] M. Lyu and Y. He. Improving the N-Version Programming Process Through the Evolution of a Design Paradigm. *Proceedings of IEEE Transactions on Reliability*, 1993.

[9] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electronic Lecture Notes in Theoretical Computer Science*, 89(2), 2003.

[10] R. Pender. Pender Electonic Design. www.pender.ch.

[11] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 110–121. ACM Press, 2003.

[12] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI*, pages 303–316, 2004.

[13] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[14] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *International Symposium on Computer Architecture (ISCA)*, pages 1–24, 2000.

[15] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Transactions on Computer Systems*, 23(1):77–110, 2005.

[16] R. Teodorescu and J. Torrellas. Prototyping Architectural Support for Program Rollback Using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2005.

[17] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *International Symposium on Computer Architecture (ISCA)*, pages 122–135. ACM Press, 2003.

[18] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, pages 269–280. IEEE Computer Society, 2004.

[19] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 224–237, June 2004.