PROTOTYPING ARCHITECTURAL SUPPORT FOR PROGRAM ROLLBACK USING PROGRAMMABLE LOGIC

BY

MIRCEA-RADU TEODORESCU

Diploma Engineer, Technical University of Cluj-Napoca, Romania, 2002

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

This thesis presents a processor and memory-hierarchy prototype based on FPGAs that provides hardware support for program rollback. We use this prototype to demonstrate how compiler- or user-controlled speculative execution can help in debugging production codes. The system is based on a synthesizable VHDL implementation of a 32-bit processor compliant with the SPARC V8 architecture. We conduct experiments on applications with real bugs. The applications run on top of a version of Linux ported to this hardware. Our experiments show that our system is able to successfully execute the buggy code sections speculatively. This allows the thorough characterization of the faulty code through repeated rollback and re-execution. Moreover, the hardware extensions we made to the baseline system increase the hardware resource requirements by less than 4.5%.

Acknowledgments

I would like to thank my adviser, Josep Torrellas, for his technical, moral and financial support. I would also like to thank the IACOMA and PROBE research groups at University of Illinois for valuable discussions and feedback on our research.

Table of Contents

List of Figures

Chapter 1

Introduction

Several recently-proposed techniques in computer architecture require speculation over long program sections. Examples of such techniques are thread-level speculation [6, 8, 22, 23], speculation on synchronization [12, 19], speculation on the values of invalidated cache lines [7], speculation on conforming to a memory consistency model [5], and speculation on the lack of software bugs [14, 28].

In all these cases, when speculation fails, the architecture has to provide a means to quickly and cleanly roll back the side effects of the speculative code. Specifically, as a thread executes speculatively, the processor buffers the register and memory state that it generates. If and when the speculation is proved to be correct, the processor commits the speculative state. If, instead, the speculation is incorrect, the state is discarded and the program execution is rolled back to the state prior to the speculative execution.

This paper describes a processor and memory-hierarchy prototype based on FPGAs that implements hardware for rollback of very long, misspeculated code sections. The prototype implements register checkpointing and restoration, buffering in the L1 cache of the state generated by retired speculative instructions, and instructions for transitioning between speculative and non-speculative execution modes.

We use the prototype to demonstrate how application rollback can help debug *production* code. The compiler inserts hints into the application to indicate regions of code that are "at risk". These suspicious regions are then executed speculatively. If an external checker detects a bug, the suspicious region is rolled back and re-executed. Upon re-execution, the software can choose to enable more instrumentation that will help characterize the buggy code region thoroughly.

For our prototype, we modify a synthesizable VHDL implementation of a 32-bit processor compliant with the SPARC V8 architecture. We map the modified processor to a Xilinx Virtex-II FPGA chip on a dedicated development board. We run several applications on top of a version of Linux running on this hardware. We choose FPGA as a target technology because it is ideal for rapid prototyping and allows us to both validate our design choices and experiment with realistic workloads.

Our measurements show that the hardware extensions required to support the rollback of very long, misspeculated code sections increase the resource requirements of the processor, when targeting FPGA technology, by less than 4.5%.

We envision this hardware as part of a larger infrastructure that includes compiler and operating system assistance for bug detection and characterization in production code.

1.1 Contributions

We extend an existing processor to include support for rapid rollback and re-execution of very large, misspeculated code sections. The extensions include cache support for holding speculative data, register checkpointing, and Instruction Set Architecture (ISA) support for compiler-directed transitions between speculative and non-speculative execution.

We prove that, with relatively simple hardware, we can provide powerful debugging support that the compiler or programmer can exploit to enable lightweight, on-the-fly debugging of production code.

We test the system on a real hardware platform based on FPGA technology. We experiment with several buggy applications running on top of a version of Linux.

Chapter 2

An integrated debugging system

The hardware that we present in this work is part of a larger debugging infrastructure that targets bug detection, characterization, and recovery for production code. This system will eventually include hardware, compiler and operating system support. Our work is focused on the hardware support, but for clarity, we give a brief description of the entire infrastructure.

In our system, a program executes in one of three states: *normal*, *speculative*, and *re-execute*. In normal mode, only minimal checking for bugs takes place; in speculative mode, the program is in a potentially buggy section of code that the hardware can roll back and re-execute. The program enters re-execution mode when a rollback has been induced. In this mode, a bug can be characterized thoroughly, through repeated rollback and re-execution, by enabling instrumentation within the application.

The transition between execution states is currently done at *Observation Points* (OP) inserted in the code by the compiler. A transition occurs as a result of a test on the program state or other external input. When a transition occurs, the hardware performs the necessary actions to enable/disable checkpointing and rollback.

2.1 Hardware support

We implemented some of the hardware support needed for thread-level speculation [6, 8, 22, 23] in a fully synthesizable system. This support includes the ability to roll back and re-execute instructions long after they have been retired. This is essential for making our desired type of speculative execution possible.

When executing in speculative mode, instructions are not allowed to change the content of main memory. All speculative data is marked and kept in the cache. It can be invalidated if necessary. The idea is to use this support as a primitive for fast and lightweight software debugging. More details about the hardware support are given in Section 4.

Our system is meant to help characterize buggy sections of code to facilitate bug detection and correction. We still need a mechanism to help us determine that a potential anomaly has occurred. In our experiments, we assume the existence of a bug detection framework similar to iWatcher [28] — 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.

2.2 Compiler support

We use a compiler [9] to detect potential anomalies in an application, and generate code necessary for the OPs. An OP consists of a test and actions. The test is used to determine when the actions should be performed. The actions include emitting information about the program state or performing execution mode transitions.

The compiler also uses heuristics to detect regions of code that should be executed in speculative mode. For a bug to be characterized, it is important that the regions of code that can lead to errors be identified.

When a potential bug has been found, the segment of code containing the error is rolled back and re-executed. Upon re-execution, instrumentation that was previously inserted by the compiler is turned on and used to characterize that code section. It is the compiler's job to determine what information is relevant, and to generate the code needed to collect it.

2.3 Operating system support

OS support is also important for bug characterization, state recovery and re-execution. The ability of the hardware to buffer speculative state is limited to instructions that touch data that can be kept in the cache. If an I/O or a non-cacheable operation is performed, the speculative execution has to be terminated, because such an instruction cannot be undone.

We envision the OS to take over in such a case and buffer the speculative state in software. This would be more costly, but would extend the speculative code section significantly. Moreover, to support bug characterization, OS support is needed to deterministically replay system events such as incoming messages. If the speculative execution section is too long and the cache is about to overflow the speculative data, the OS can again be invoked to buffer the speculative state by using a mechanism like copy-on-write.

Chapter 3

Applications of program execution rollback

The architectural support presented in this work provides a flexible environment that can be used to support a wide range of techniques that address system reliability in general. In this section, we describe some of these applications.

3.1 Code versioning

Code versioning, or N-version programming [11] 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 function includes some verification code that checks whether the computation was correct or not. We 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 less likely to both fail.

3.2 OS kernel and driver debugging

One of the major challenges in OS reliability is ensuring the correct functioning 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 [2]. Several solutions have been proposed to this problem and the majority of them involve isolating the kernel from the device drivers with some protection layer [25]. 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, any function calls from kernel to driver or vice-versa could be executed speculatively. If an error is detected, the changes made to kernel memory could 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 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.

3.3 Lightweight information collection and sampling

Detecting bugs in production code can be challenging because it is hard to obtain information about program execution. It is hard to collect relevant information without incurring a large overhead. Previous solutions [10] to this problem have suggested using statistical sampling to obtain execution information with small overheads.

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 a 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 low.

3.4 Failure-oblivious computing

A failure-oblivious system [20] enables programs to continue executing through memory errors without memory corruption. 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.

3.5 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.

Chapter 4

Implementation

As a base for our implementation, we used a synthesizable VHDL implementation of a 32-bit processor [4] compliant with the SPARC V8 architecture. This implementation has an in-order, single-issue, five stage pipeline and one level of instruction and data caches. It has a hardware multiplier and divider, an interrupt controller and two UART units. The processor implements a windowed register file with a variable number of windows. It is part of a system-on-a-chip infrastructure that includes a synthesizable SDRAM controller, PCI and Ethernet interfaces.

In order to support lightweight rollback an replay over relatively long code sections, we need to implement two main extensions to the existing system: (1) a cache that buffers speculative data and supports rollback and (2) register checkpointing and rollback. This allows speculative instructions to retire by storing the speculative data they generate into the cache and ensures that the register state of the processor before a checkpoint can be restored in case of a rollback request. We now describe both extensions in some detail. We also show how the transitions between execution modes are controlled by software.

4.1 Data cache with rollback support

In order to allow the rollback of speculative instructions, we need to make sure that the data they generate can be invalidated if necessary. To this end, we keep the speculative data (the data generated by the system while executing in speculative mode) in the cache, and do not allow it to change the memory state. To avoid a costly cache flush when transitioning between execution modes, the cache must be able to hold both speculative and non-speculative data at the same time. For this, we use a cache designed to store multiple versions of data. This is done by adding a version identifier to each cache line. Two versions (represented by one version bit per cache line) are sufficient. Version 0 corresponds to non-speculative, and version 1 to speculative state.

In addition to the version bit, we extended the cache controller with a Cache Walk State Machine (CWSM) that is responsible for traversing the cache and clearing the version bit (in the case of a successful commit) or invalidating the speculative lines (in case of rollback).

The version bit is stored at line granularity. Therefore, one cache line can hold only one version of data at a time. For this reason, while the processor is in speculative mode, for every write hit we check if the line we are writing to contains non-speculative, dirty data. If it does, we write-back the dirty data, update the line, and then set the version bit to speculative. From this point on, the line is speculative and will be invalidated in case of a rollback.

While in speculative mode, if a line is about to be evicted, we first check if it is speculative. If it is, we choose a non-speculative line in the same set for eviction. If one does not exist, we must end the speculative section and commit.

4.1.1 The Cache Walk State Machine

The Cache Walk State Machine (CWSM) is used to traverse the entire data cache and either commit or invalidate the speculative data. The state machine is activated when a commit or rollback instruction reaches the Memory stage of the pipeline. The pipeline is stalled and the cache controller transfers control to the CWSM. The CWSM has three states as shown in Figure 4.1.

In case of commit, the CWSM uses the Walk state to traverse the cache and clear the

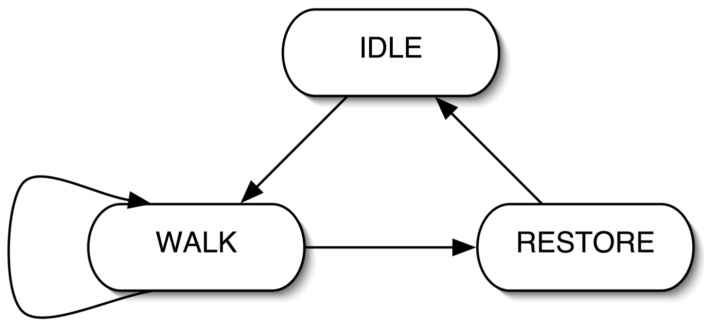

Figure 4.1: Cache Walk State Machine.

version bits, effectively merging the speculative and non-speculative data. The traversal takes one cycle for each line in the cache. In the case of rollback, the CWSM is called to invalidate all the speculative lines in the cache. This means traversing the cache and checking the version bit for each line. If the line contains speculative data, the version and valid bits are cleared. The Restore state is used to restore the controller to the initial state and release the pipeline.

4.1.2 Technology constraints

Some of the design decisions we made were influenced by the target technology chosen for our implementation (Xilinx Virtex II family of FPGAs). The cache is implemented with synchronous RAM blocks present in the FPGA chip. This allows the cache to be quite fast, with a single-cycle access time.

On the other hand, a disadvantage of using these memory structures is that they cannot be modified to incorporate additional control signals. For instance we would have liked to use a *clear_all* signal for the version bit. This would have allowed a single-cycle "one-shot" clear of all version bits and thus a single-cycle transition from speculative to non-speculative execution in the commit scenario.

4.2 Register checkpointing and rollback

Before transitioning to speculative state, we must ensure that the processor can be rolled back to the current, non-speculative state. The current state includes the processor status registers, global registers, register file and the data cache. The data cache rollback is accomplished through versioning as described in the previous section. For the register file, we checkpoint it when we enter the speculative section and restore it if we need to roll back. Register file checkpointing and rollback can be performed either in software or in hardware. In the software approach, the compiler inserts explicit store instructions to save to memory all the variables that are currently in registers. This software checkpoint would have to be included in all OPs that can cause a transition to speculative mode. This can be costly in terms of performance and can lead to significant code expansion.

The problem is worse in the case of a SPARC V8 processor because it implements a windowed register file (WRF). At any one time during execution, a program sees 8 global registers plus a 24-register window within a larger register file. On a procedure call, instead of saving local registers on the stack, the current window is simply shifted. A new set of registers is available to the callee. Upon return from the procedure call, the window is shifted back and the old registers become available.

At any time, a large number of variables can be in the register file. In order to checkpoint the state of the processor, the entire valid content of the WRF must be saved, not just the current window (in the worst case, the entire register file). If performed in software, this can be very expensive, since the SPARC V8 architecture specifies a limit of up to 520 registers for its WRF!

For this reason, we perform the register checkpointing in hardware. This is done using a Shadow Register File (SRF), a memory structure identical to the main register file. Before entering speculative execution, the pipeline is notified that a checkpoint needs to be taken. The pipeline stalls and control is passed to the Register Checkpointing State Machine

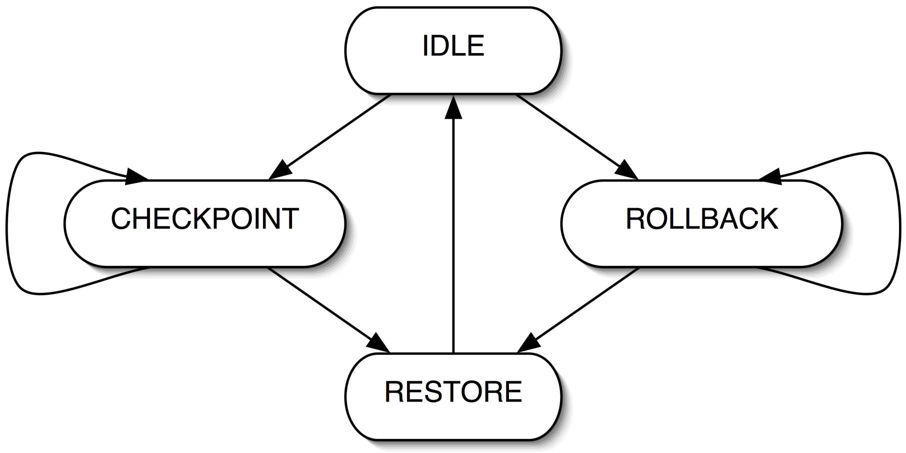

Figure 4.2: Register Checkpointing State Machine.

(RCSM). The RCSM has four states and is responsible for coordinating the checkpoint as shown in Figure 4.2.

The RCSM is in the Idle state while the pipeline is executing normally. A transition to the Checkpoint state occurs before the processor moves to speculative mode. While in this state, the valid registers in the main register file are copied to the SRF. The register file is implemented in SRAM and has two read ports and one write port. This means that we can only copy one register per cycle. Thus the checkpoint stage takes as many cycles as there are valid registers in the register file plus one cycle for all the status, control and global registers (these are not included in the same memory structure and can all be copied in one cycle).

The Rollback state is activated when the pipeline receives a rollback signal. While in this state, the contents of the register file is restored from the checkpoint, along with the status and global registers. Similarly, this takes as many cycles as there are valid registers.

4.3 Changing the execution mode

4.3.1 Enabling speculative execution

The transition to speculative execution is triggered by a LDA (Load Word from Alternate Space) instruction with a dedicated ASI (Address Space Identifier). These are instructions introduced in the SPARC architecture to give special access to memory (for instance, access to the tag memory of the cache). We extended the address space of these instructions to give us software control over the speculative execution.

The special load is allowed to reach the Memory stage of the pipeline. The cache controller detects, initializes and coordinates the transition to speculative execution. This is done at this stage rather than at Decode because, at this point, all non-speculative instructions have been committed or are about to finish the Write Back stage. This means that, from this point on, any data written to registers or to the data cache is speculative and can be marked as such.

The cache controller signals the pipeline to start register checkpointing. Interrupts are disabled to prevent any OS intervention while checkpointing is in progress. Control is transferred to the RCSM, which is responsible for saving the processor status registers, the global registers, and the used part of register file.

When this is finished, the pipeline sends a *checkpointing complete* signal to the cache controller. The cache controller sets its state to speculative. Next, the pipeline is released and execution resumes. From this point on, any new data written to the cache is marked as speculative.

4.3.2 Exiting speculative execution

Speculative execution can be ended either explicitly by an instruction or implicitly by an event that cannot be rolled back.

Normally, speculative execution ends with commit, which merges the speculative and non-speculative states. On the other hand, if a bug is detected, speculation ends by triggering a rollback.

Both cases are triggered by a LDA instruction with a dedicated ASI. The distinction between the two is made through the value stored in the address register of the load instruction.

An LDA from address 0 causes a commit. In this case, the pipeline allows the load to reach the Memory stage. At that point, the cache controller takes over, stalls the pipeline, and passes control to the CWSM. The CWSM is responsible for traversing the cache and resetting the version bit. When the cache walk is complete, the pipeline is released and execution can continue non-speculatively.

An LDA from any other address triggers a rollback. When the load reaches the Memory stage, the cache controller stalls the pipeline and control goes to the RCSM. The register file, global and status registers are restored. The nextPC is set to the saved PC. A signal is sent to the cache controller when rollback is done. At the same time, the cache controller uses the CWSM to traverse the cache, invalidating speculative lines and resetting the version bits. When both the register restore and cache invalidation are done, the execution can resume.

The value passed to the LDA instruction can be set dynamically, based on some event that can help determine whether a problem might have occurred.

4.4 Speculative window size

The number of instructions that are successfully rolled back is ideally given by the distance between a begin speculation and an end speculation instruction. We call this a speculative window. There are, however, two events that can force the premature end of a speculative section: cache overflow and I/O access.

A cache overflow occurs when a line needs to be displaced form a cache set and all the lines in the set are speculative. This means that speculative data can no longer be held in the cache.

I/O operations are a major concern in rollback/replay systems because they cannot be undone. They are identified by the cache controller which conservatively considers all non-cacheable memory accesses as I/O accesses.

In both situations the OS is informed about the exceptional condition by raising an exception. The exception handler can take a variety of actions. For instance it could save the speculative data in memory or record I/O operations. For simplicity, our prototype currently triggers an early commit of the speculative section.

Overall, the size of the code that can be executed speculatively is dependent on a variety of factors. Some are application-related, such as: frequency of the I/O accesses, memory footprint size and access pattern or interaction with the OS. Other factors are strictly related to hardware resources such as cache size and associativity.

4.5 Performance monitoring

In addition to exposing control over the speculative execution to the software, we provide some feedback on the state of the processor while in speculative mode. This information can be used to fine-tune the instrumentation and can help with debugging.

We introduce an LDA instruction that can be used to probe the state of the processor. Based on its return value, we determine if the processor is in normal, speculative, or re-execute mode (after a rollback). This can be very useful if we want to execute code selectively, based on the state of the processor.

The end-speculation instruction provides additional information on the speculation outcome. It returns 0 if the speculative execution ended normally (with commit or rollback), and a non-zero value if some event forced an early commit. The value returned in this case represents the event that caused the early commit.

We also implemented a counter that keeps track of how many dynamic instructions are

```
enter_spec(){            /* Begin speculation */
 asm(" stbar             /* fence */
     mov 0, %o0
     lda [%o0] 0x8, %o1
     nop ");
}

exit_spec(int flag){     /* End speculation */
 asm(" stbar             /* fence */
     mov flag, %o0       /* if flag=0 commit */
     lda [%o0] 0x9, %o1  /* else rollback */
     nop ");
}
```

Figure 4.3: Begin and end speculation instructions.

executed speculatively. The counter is stopped when speculative execution ends, and can be read with a special LDA instruction.

4.6 Using program rollback for debugging

Finding bugs in software requires gathering as much information as possible about the circumstances in which bugs occur. We provide a mechanism for the compiler or the programmer to execute sections of code speculatively. We rely on an external detection mechanism to identify a possible problem and trigger a rollback. Upon re-execution, more instrumentation can be turned on to characterize that section of code.

We define two functions, namely `enter_spec()` used to begin speculative execution, and `exit_spec()` to end speculative execution with commit or rollback. `exit_spec()` takes one argument, `flag`, which indicates whether speculation ends with commit or rollback. If a bug has been detected by some external mechanism, the `flag` variable is set to some non-zero value, and a rollback is triggered at `exit_spec()`. The following code shows the implementation.

We define a function `proc_state()` to probe the state of the processor as detailed in

```
num=1;                  /* non-speculative code */
...
enter_spec();           /* begin speculation */
...
p=m[a[*x]]+&y;          /* pointer arithmetic */
...
if (bug_suspected)
 flag=1;
...
if (proc_state()==2) {  /* info collection */
 info_collect();        /* only in re-execute mode */
}
...
exit_spec(flag);        /* end speculation */
num++;                  /* non-speculative code */
...
```

Figure 4.4: Code instrumentation example.

Section 4.5. The return value 0 means normal mode, 1 speculative mode and 2 represents the re-execute mode. The following code shows how these functions can be used to characterize a section of buggy code.

The compiler or the programmer identifies regions of code that are "at risk". Using the begin/end speculation pair of instructions, that section of code can be executed speculatively. If a bug is suspected, the program sets `flag`, and when `exit_spec()` is executed, a rollback is triggered. The execution resumes from the `enter_spec()` instruction and the code is re-executed.

The compiler can also insert 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, when a potential problem has been found, so it does not introduce significant overhead on correct runs.

Figure 4.5 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.

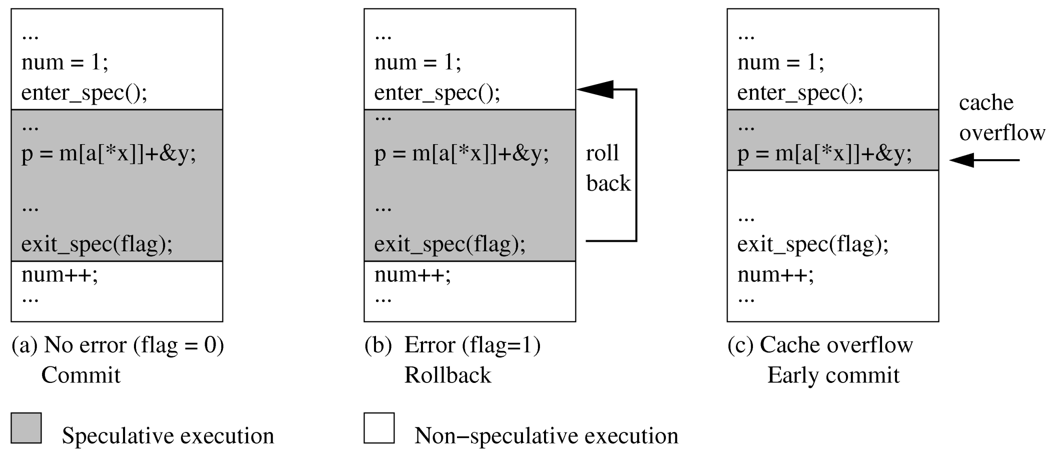

Figure 4.5: Three possible execution scenarios for speculative code.

In case (b), an abnormal behavior that can lead to a bug is encountered. `Flag` is set by the program and when execution reaches `exit_spec(flag)` the execution rolls back to the beginning of the speculative region. This can be repeated 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 exception handler decides to commit the current speculative data and continue executing normally. When the execution reaches the `exit_spec(flag)` instruction, the state of the processor is first checked. Since the processor is no longer speculative (due to the early commit), the instruction is simply ignored and execution continues normally.

Chapter 5

Evaluation

5.1 Experimental infrastructure

5.1.1 FPGA system

As a platform for our experiments, we used LEON2 [4], a synthesizable VHDL implementation of a 32-bit processor compliant with the SPARC V8 architecture.

The processor has an in-order, single-issue, five stage pipeline (Fetch, Decode, Execute, Memory and Write Back). Most instructions take 5 cycles to complete if no stalls occur. The Decode and Execute stages are multi-cycle and can take up to 3 cycles each.

The data cache can be configured as direct mapped or as multi-set with associativity of up to 4, implementing least-recently used (LRU) replacement policy. The set size is configurable to 1-64 KBytes and divided into cache lines of 16-32 bytes. Each line has a tag field, and valid and dirty bits for each 4-byte sub-block. The per-word valid bits allow partially valid lines to exist in the cache. On a data cache read miss, only 4 bytes of data are loaded into the cache from main memory. This reduces the number of ports (because no additional write ports are needed for line refill) and eliminates the need for refill logic. We implemented a write-back cache controller since the initial system had a write-through data cache. The data cache needs to be write-back to make holding speculative data possible.

This processor is part of a system-on-a-chip infrastructure that includes a synthesizable

SDRAM controller, PCI and Ethernet interfaces. The system is 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 [16]. The board has 8MB of FLASH PROM and 64 MB SDRAM. Communication with the device, loading of programs in memory, and control of the development board are all done through the PCI interface from a host computer. Console output is sent on the serial interface.

5.1.2 Operating system

On this hardware we run a special version of the SnapGear Embedded Linux distribution [3]. 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 applications.

5.1.3 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 mechanism like iWatcher [28] (see Section 2.1).

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

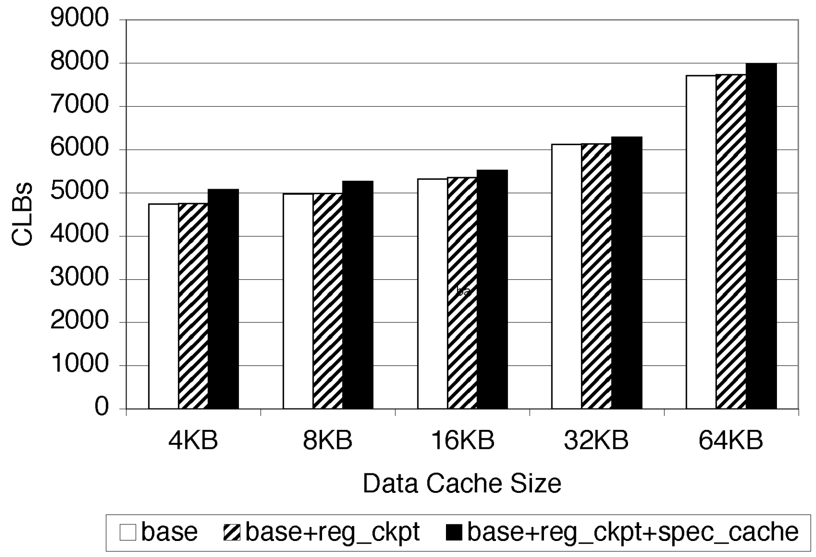

Figure 5.1: CLBs utilization.

experiments are done under realistic conditions with the applications running on top of Linux.

5.2 Results

5.2.1 Hardware overhead

To get a sense of the hardware overhead imposed by our program rollback support, we synthesize just the processor core (including the cache but not the memory, PCI or serial controllers). We look at the utilization of two main resources: Configurable Logic Blocks (CLBs) and SelectRAM memory blocks.

The Virtex II CLBs are organized in an array and are used to build the combinational and synchronous logic components of the design. Each CLB element is tied to a switch matrix to access the general routing matrix. A CLB element comprises 4 similar slices. Each slice includes two 4-input function generators, carry logic, arithmetic logic gates, wide-function multiplexers and two storage elements. Each 4-input function generator is programmable

Processor	LEON2, SPARC V8 compliant
Clock frequency	40MHz
Instruction cache	8KB
Data cache	32KB
Main memory	64MB
Windowed register file	8 windows × 24 registers
Global registers	8 registers

Table 5.1: Main parameters of the experimental system.

as a 4-input lookup table (LUT), 16 bits of distributed SelectRAM memory, or a 16-bit variable-tap shift register element.

The SelectRAM memory blocks are 18 Kbit, dual-port RAMs with two independently-clocked and independently-controlled synchronous ports that access a common storage area. Both ports are functionally identical. The SelectRAM block supports various configurations, including single- and dual-port RAM and various data/address aspect ratios. These devices are used to implement the large memory structures in our system (data and instruction caches, the register file, shadow register file, etc).

Figure 5.1 shows a comparison between the number of CLBs used for three configurations of the processor core and five different sizes of the data cache. The *base* represents the original processor core, the *base+reg_ckpt* represents the original processor plus the register checkpointing mechanism and finally, the *base+reg_ckpt+spec_cache* represents the system with both register checkpointing and data cache support for speculation. As we can see, the CLB overhead of adding program rollback support in hardware is small (less than 4.5% on average) and relatively constant across the range of cache sizes that we tested. One thing we can notice is that the hardware overhead introduced by the register checkpointing additions is very small compared to the cache overhead. This is most likely due to a simpler design and smaller number of control signals necessary for the RCSM.

Figure 5.2 shows a comparison between the same configurations, but looking at the number of SelectRAM blocks utilized. Again, the amount of extra storage space necessary

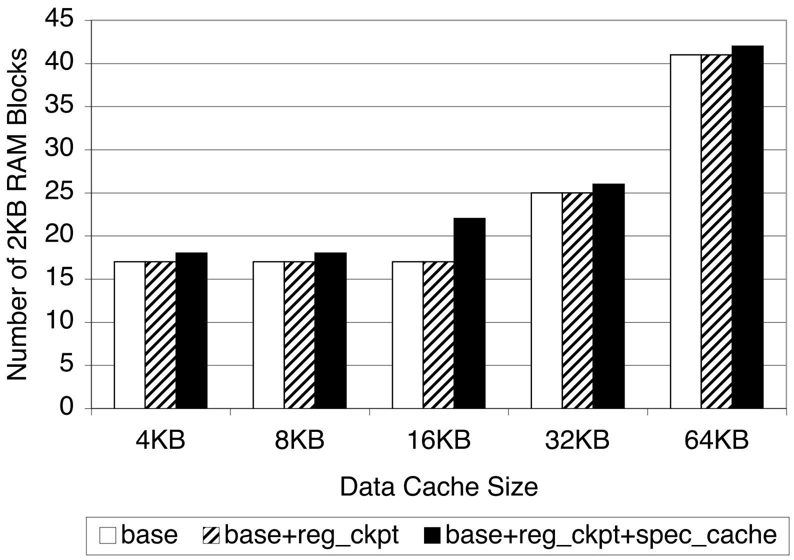

Figure 5.2: RAM utilization.

for our system is small across the five configurations that we evaluated.

5.2.2 Speculative execution of buggy applications

We run experiments on the buggy programs to evaluate the behavior of the system in the presence of a few types of memory related bugs. Details about the experimental setup are given in Table 5.2.1.

We manually instrument the code with the instructions that enable and disable speculative execution. Normally, this would be done by the compiler using profiling information or other heuristics to determine which sections of code should be monitored. We assume the existence of an anomaly-detection mechanism such as iWatcher [28]. We want to determine if we can speculatively execute the section of dynamic code that contains both the bug and the detection location. This will allow the rollback and re-execution of the buggy code section in order to characterize the bug thoroughly by enabling additional instrumentation.

Table 5.2.1 shows that the buggy sections were successfully rolled back in most cases, as shown in column four. That means that the system speculatively executed the entire section

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, 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 5.2: Speculative execution in the presence of bugs.

from when the bug occurs to when the bug is detected, then reached the end-speculation instruction, and rolled back. On the other hand, a failed rollback means that, before reaching the end-speculation instruction, a cache overflow occurs, which 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 are executed speculatively within a single speculative window that contains both the bug and detection location. Notice that in the case of *polymorph* the large number of dynamic instructions cause the cache to overflow the speculative data, and force an early commit.

Chapter 6

Related work

Some of the hardware presented in this work builds on extensive work on thread-level speculation (TLS) (e.g., [6, 8, 22, 24, 23]). We employ some of the techniques first proposed for TLS to provide lightweight rollback/replay capabilities. TLS hardware has also been proposed as a mechanism to detect data races on-line [18].

Previous work has also focused on various methods for collecting information about bugs. The "Flight Data Recorder" [26] enables off-line deterministic replay of applications and can be used for postmortem analysis of a bug.

There is other extensive work in the field of software-based dynamic execution monitoring. Well-known examples include Eraser [21], Valgrind [13] and others [1, 15, 17, 10]. 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. These systems have overheads that are typically too large to make them acceptable in production code.

There have also been proposals for hardware support for debugging such as iWatcher [28] and AccMon [27]. 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.

Chapter 7

Conclusions

The work presented here shows that with relatively simple hardware we can provide powerful support for debugging production codes. We build a hardware prototype of the envisioned system, using FPGA technology. We run experiments on top of Linux running on this system.

The hardware presented in this work is part of a comprehensive debugging infrastructure that also includes compiler and OS support. The compiler identifies vulnerable code regions as well as instruments the code with speculation control instructions. The OS assists in extending the speculative window beyond the limits of the cache.

The FPGA technology proved to be a very appropriate target for our prototype. It allows us to validate our design and experiment with realistic workloads. We are able to look at issues related to hardware and operating system interaction necessary for developing a system-wide approach to the debugging challenge.

We are also investigating other applications of processor execution rollback. We are looking at ways in which our architecture can improve techniques like N-version programming for reliability and performance, non-intrusive, low-overhead fault injection into long-running applications, and resilience to transient faults.

References

[1] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[2] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. *SIGOPS Operating Systems Review*, 35(5):73–88, 2001.

[3] CyberGuard. Snapgear embedded linux distribution. www.snapgear.org.

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

[5] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171. IEEE Computer Society, 1999.

[6] Lance Hammond, Mark Willey, and Kunle Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[7] Jaehyuk Huh, Jichuan Chang, Doug Burger, and Gurindar S. Sohi. Coherence decoupling: Making use of incoherence. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–106. ACM Press, 2004.

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

[9] Sang Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *Languages and Compilers for Parallel Computing (LCPC)*, pages 539–553, 2003.

[10] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Not.*, 38(5):141–154, 2003.

[11] 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.

[12] Jose F. Martínez and Josep Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 18–29. ACM Press, 2002.

[13] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Lecture Notes in Theoretical Computer Science*, 89(2), 2003.

[14] Jeffrey Oplinger and Monica S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 184–196, October 2002.

[15] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software Practice and Experience*, 27(1):87–110, 1997.

[16] Richard Pender. Pender Electonic Design. www.pender.ch.

[17] Dejan Perkovic and Peter J. Keleher. A Protocol-Centric Approach to on-the-Fly Race Detection. *IEEE Transactions on Parallel Distributed Systems*, 11(10):1058–1072, 2000.

[18] Milos Prvulovic and Josep 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.

[19] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, pages 294–305, Austin, TX, December 2001.

[20] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[21] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[22] G.S. Sohi, S.E. Breach, and T.N. Vijayakumar. Multiscalar Processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[23] J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *High Performance Computer Architecture (HPCA)*, February 1998.

[24] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *International Symposium on Computer Architecture (ISCA)*, pages 1–24, 2000.

[25] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, 2005.

[26] Min Xu, Rastislav Bodik, and Mark 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.

[27] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep 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.

[28] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architecture Support for Software Debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 224–237, June 2004.