The Design Complexity of Program Undo Support in a General-Purpose Processor

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

1 Introduction

Several recently-proposed architectural techniques require speculation over long program sections. Examples of such techniques are thread-level speculation [4, 6, 11, 12], speculation on collision-free synchronization [7, 10], speculation on the values of invalidated cache lines [5], speculation on conforming to a memory consistency model [3], and even speculation on the lack of software bugs [8, 13].

In these techniques, as a thread executes speculatively, the architecture has to buffer the memory state that the thread is generating. Such state can potentially be quite large. If the speculation is shown to be correct, the architecture commits the speculative state. If, instead, the speculation is shown to be incorrect, the speculative state is discarded and the program is rolled back to the beginning of the speculative execution.

A common way to support these operations with low overhead is to take a checkpoint when entering speculative execution and buffer the speculative state in the cache. If the speculation is shown to be correct, the state in the cache is merged with the rest of the program state. If the speculation is shown to be incorrect, the speculative state buffered in the cache is invalidated and the register checkpoint is restored.

While the hardware needed for these operations has been discussed in many papers, it has not been implemented before. In fact, there is some concern that the hardware complexity may be too high to be cost effective.

In this paper, we set out to build such architectural support on a simple processor and prototype it using FPGA (Field Programmable Gate Array) technology. The prototype implements register checkpointing and restoration, speculative state buffering in the L1 cache for later commit or discarding, and instructions for transitioning between speculative and non-speculative execution modes. The result is a processor that can cleanly roll back (or "undo") a long section of a program.

We estimate the design complexity of adding the hardware support for speculative execution and rollback using three metrics. The first one is the hardware overhead in terms of logic blocks and memory structures. The second one is development time, measured as the time spent designing, implementing and testing the hardware extensions that we add. Finally, the third metric is the number of lines of VHDL code used to implement these extensions.

For our prototype, we modified LEON2 [2], a synthesizable VHDL implementation of a 32-bit processor compliant with the SPARC V8 architecture. We mapped the modified processor to a Xilinx Virtex-II FPGA chip on a dedicated development board. This allowed us to run several applications, including a version of Linux.

Our measurements show that the complexity of supporting program rollback over long code sections is very modest. The hardware required amounts to an average of less than 4.5% of the logic blocks in the simple processor analyzed. Moreover, the time spent designing, implementing, and debugging the hardware support is only about 20% higher than adding write back support to a write-through cache. Finally, the VHDL code written to implement our hardware adds about 14.5% more code to the data cache controller, and 7.5% to the simple pipeline.

This paper is organized as follows: Section 2 outlines the implementation; Section 3 estimates the complexity in terms of hardware overhead, rough development time, and lines of code; and Section 4 concludes.

2 Implementation

In order to support lightweight rollback and replay over relatively long code sections, we need to implement two main extensions to a simple processor: (1) modify the cache to also buffer speculative data and support rollback and (2) add support for register checkpointing and rollback. This allows a retiring speculative instruction to store the speculative datum that it generates into the cache, and ensures that the register state of the processor before speculation can be restored in case of a rollback request. We now describe both extensions in some detail. We also show how the transitions between non-speculative and speculative execution modes are controlled by software.

2.1 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 add a single *Speculative* bit per cache line. If the Speculative bit is 0, the line does not contain speculative data. Otherwise, the line contains speculative data, and the non-speculative version of the line is in memory.

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

The Speculative bit is stored at line granularity. Therefore, 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 Speculative bit. From this point on, the line is speculative and will be invalidated in case of a rollback.

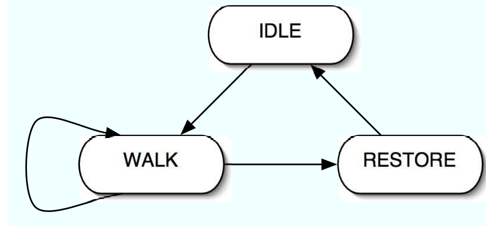

Figure 1. Cache Walk State Machine.

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 none exists, we end the speculative section. In this initial version of our prototype, we commit the section at this point.

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 (Section 2.3) 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 1.

In case of commit, the CWSM uses the Walk state to traverse the cache and clear the Speculative 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 Speculative bit for each line. If the line contains speculative data, the Speculative and Valid bits are cleared.

2.2 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. Consequently, we checkpoint the register file. This is done using a Shadow Register File (SRF), a 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 Regis-

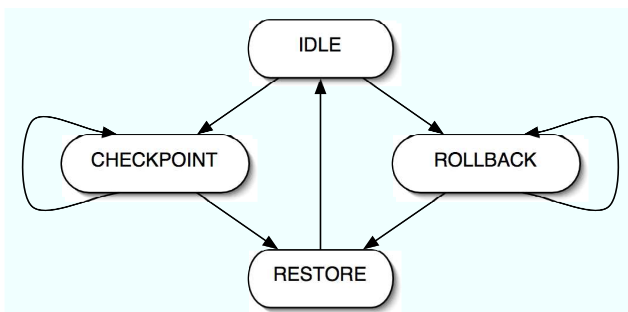

Figure 2. Register Checkpointing State Machine.

ter Checkpointing State Machine (RCSM). The RCSM has four states as shown in Figure 2, and is responsible for coordinating the checkpoint.

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. In addition, we need one cycle for all the status, control and global registers. These are not included in the SRF 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, plus one.

2.3 Controlling speculative execution

There are several possible approaches to control when to enter and exit speculative execution. One approach is to have instructions that explicitly mark the beginning and end of speculative execution. A second approach is to use certain hardware events to trigger the beginning and the end of speculative execution. Finally, it is possible to implicitly keep the processor always in speculative mode, by using a hardware-managed "sliding window" of speculative instructions. In this prototype, we have implemented the first approach.

2.3.1 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.

2.3.2 Exiting speculative execution

Speculative execution can end either with a commit, which merges the speculative and non-speculative states or with a rollback in case some event that requires an "undo" is encountered. 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 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 Speculative 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 speculative bits. When both the register restore and cache invalidation are done, the execution can resume.

3 Evaluation

3.1 Experimental infrastructure

As a platform for our experiments, we used LEON2 [2], 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. The 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 [9]. 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.

On this hardware we run a special version of the SnapGear Embedded Linux distribution [1]. SnapGear Linux is a full source package, containing kernel, libraries and application code for rapid development of embedded Linux systems. A cross-compilation toolchain for the SPARC architecture is used for the compilation of the kernel and applications.

3.2 Estimating design complexity

We estimate the design complexity of adding our hardware extensions using three metrics: the hardware overhead in terms of logic blocks and memory structures, implementation time, and VHDL code size.

3.2.1 Hardware overhead

One approach to estimating the complexity of our design is to look at the hardware overhead imposed by our scheme. We synthesize the processor core, including the cache. 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 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).

We also measure the hardware overhead introduced by implementing a write-back cache controller. The original LEON2 processor has a write-through data cache. Since our system needs the ability to buffer speculative data in the cache, a write back cache is needed. We implement it by modifying the existing

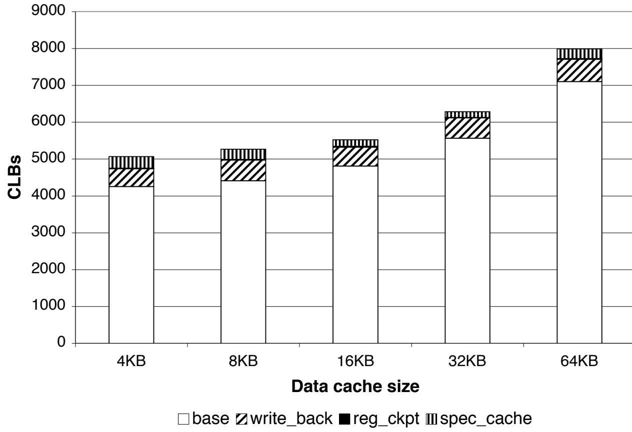

Figure 3. Number of CLBs used by different hardware structures. Each bar corresponds to a core with a different cache size.

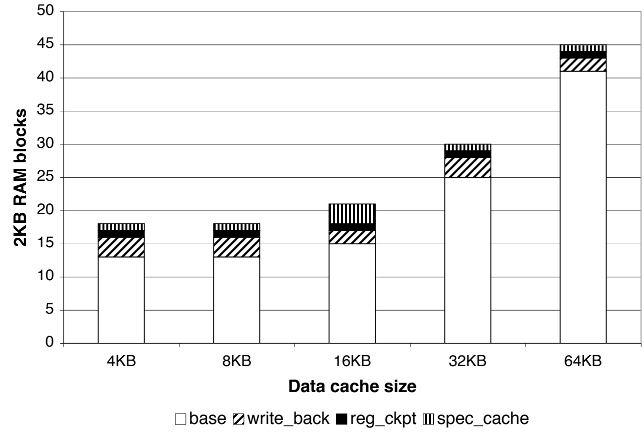

Figure 4. Number of SelectRAM blocks used by different hardware structures. Each bar corresponds to a core with a different cache size.

controller. This allows us to compare the implementation complexity of the hardware we propose with the complexity of modifying a write-through cache into a write-back one.

Figure 3 shows a breakdown of the number of CLBs used by the processor core and each of the main extensions added to it. Each bar corresponds to a core with a different data cache size. The *base* represents the size of the original processor core with a write-through cache; the *write_back* represents the overhead of adding the write back cache; the *reg_ckpt* represents the register checkpointing mechanism; and finally, the *spec_cache* represents the cache support for speculative data.

The CLB overhead of adding program rollback support (*reg_ckpt* plus *spec_cache*) to a processor is small (less than 4.5% on average) and relatively constant across the range of cache sizes that we tested. This overhead is computed with respect to the processor *with* the write back data cache controller (*base* plus *write_back*), a configuration typical for most current processors.

We notice that the hardware overhead introduced by the register checkpointing support is very small compared to the rollback support in the cache. This is due to a simple design of the register checkpointing state machine which requires less state information and fewer control signals. Also, the overhead of adding the write back cache controller is larger than that of adding the full support for speculative execution.

Figure 4 shows a comparison between the same configurations, but looking at the number of SelectRAM blocks utilized. We see that the amount of extra storage space necessary for our additions is small across the five configurations that we evaluated.

3.2.2 Design, implementation and testing time

Another complexity indicator is the time spent designing, implementing and debugging the hardware. We consider the three major components of our design, namely the speculative cache, the register checkpointing and the software control support. We compare the time spent developing them with the time spent developing the write back cache controller.

The estimates are shown in Figure 5. We note that out of the three extensions, the speculative cache took the longest to develop. Overall, the time spent designing, implementing, and debugging our hardware support is only about 20% higher than adding write back support to a write-through cache.

3.2.3 Lines of VHDL code

The third measure of complexity we use is VHDL code size. The processor is implemented in a fully synthe-

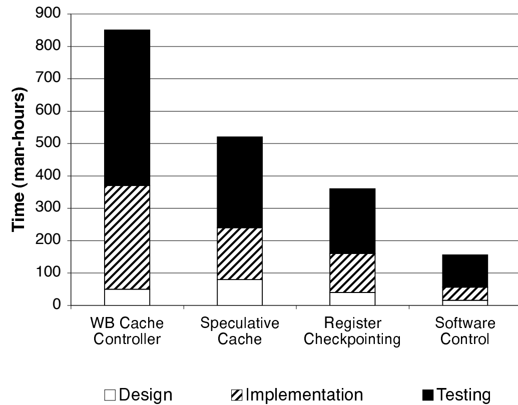

Figure 5. Estimates of the design, implementation and testing time (in man hours) for individual components in our system.

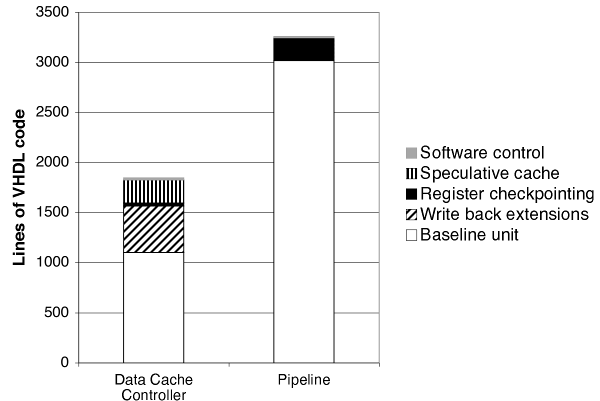

Figure 6. Breakdown of the lines of VHDL code in the data cache controller and the pipeline.

sizable RTL description that provides sufficient details to make code size a good indication of design complexity.

We look at the number of lines of VHDL code needed to implement our hardware extensions in the two main processor components that we modified: the data cache controller and the pipeline. The results are shown in Figure 6. The bars show a breakdown of the lines of code for the two components. We also include data about the write back support in the cache controller.

We note that the code needed to implement our extensions is small. The results are consistent with the previous two experiments. The write back cache controller is the most complex to implement, accounting for as much code as all the other extensions combined. The VHDL code written to implement our hardware adds about 14.5% more code to the data cache controller, and 7.5% to the simple pipeline.

4 Discussion

While our conclusions have to be qualified by the fact that we are dealing with a simple processor, our working prototype has given us good insights into the complexity of developing hardware support for program rollback.

Our analysis shows that the complexity of supporting program rollback over long code sections is very modest. The hardware required amounts to an average of less than 4.5% of the logic blocks in the simple processor analyzed. Moreover, the time spent designing, implementing, and debugging the hardware support is only about 20% higher than adding write back support to a write-through cache. Finally, the VHDL code written to implement our hardware adds about 14.5% more code to the data cache controller, and 7.5% to the pipeline.

Considering that the hardware support described can be used in many novel speculative techniques (Section 1), we argue that it is complexity effective.

References

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

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

[3] C. Gniady, B. 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.

[4] L. Hammond, M. Willey, and K. 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.

[5] J. Huh, J. Chang, D. Burger, and G. 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.

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

[7] J. F. Martínez and J. 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.

[8] J. Oplinger and M. 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, Oct. 2002.

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

[10] 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, Dec. 2001.

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

[12] 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.

[13] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. 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.