SWICH: A PROTOTYPE FOR EFFICIENT CACHE-LEVEL CHECKPOINTING AND ROLLBACK

EXISTING CACHE-LEVEL CHECKPOINTING SCHEMES DO NOT CONTINUOUSLY SUPPORT A LARGE ROLLBACK WINDOW. IMMEDIATELY AFTER A CHECKPOINT, THE NUMBER OF INSTRUCTIONS THAT THE PROCESSOR CAN UNDO FALLS TO ZERO. TO ADDRESS THIS PROBLEM, WE INTRODUCE *SWICH*, AN FPGA-BASED PROTOTYPE OF A NEW CACHE-LEVEL SCHEME THAT KEEPS TWO LIVE CHECKPOINTS AT ALL TIMES, FORMING A SLIDING ROLLBACK WINDOW THAT MAINTAINS A LARGE MINIMUM AND AVERAGE LENGTH.

Radu Teodorescu
Jun Nakano
Josep Torrellas
University of Illinois at Urbana-Champaign

Backward recovery through checkpointing and rollback is a popular approach in modern processors to providing recovery from transient and intermittent faults.[1] This approach is especially attractive when designers can implement it with very low overhead, typically using dedicated hardware support.

The many proposals for low-overhead checkpointing and rollback schemes differ in a variety of ways, including the level of the memory hierarchy checkpointed and the organization of the checkpoint relative to the active data. Schemes that checkpoint at the cache level occupy an especially attractive design point.[2,3] The hardware for these schemes, which works by regularly saving the data in the cache in a checkpoint that can later serve for fault recovery, can be integrated relatively inexpensively and used efficiently in modern processor microarchitectures.

Unfortunately, cache-level checkpointing schemes such as Carer[2] and Sequoia[3] don't continuously support a large *rollback window*—the set of instructions that the processor can undo by returning to a previous checkpoint if it detects a fault. Specifically, in these schemes, immediately after the processor takes a checkpoint, the window of code that it can roll back typically is reduced to zero.

In this article, we describe the microarchitecture of a new cache-level checkpointing scheme that designers can efficiently and inexpensively implement in modern processors and that supports a large rollback window continuously. To do this, the cache keeps data from two consecutive checkpoints (or epochs) at all times. We call this approach a *sliding* rollback window, and we've named our scheme Swich—for Sliding-Window Cache-Level Checkpointing.

We've implemented a prototype of Swich using field-programmable gate arrays (FPGAs), and our evaluation of the prototype shows that supporting cache-level check-

Published by the IEEE Computer Society

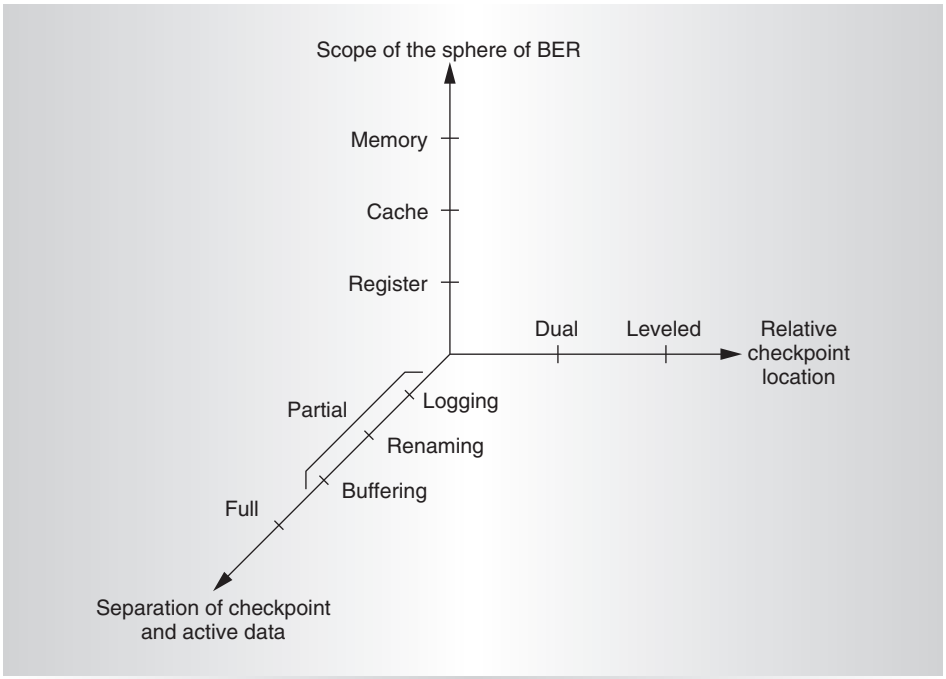

Figure 1. Taxonomy of hardware-based checkpointing and rollback schemes.

pointing with a sliding window is promising. For applications without frequent I/O or system calls, the technique sustains a large minimum rollback window and adds little additional logic and memory hardware to a simple processor.

Low-overhead checkpointing schemes

To help in situating Swich among the alternatives, it's useful to classify hardware-based checkpointing and rollback schemes into a taxonomy with three axes:

- the scope of the sphere of backward error recovery (BER),
- the relative location of the checkpoint and active data (the most current versions), and
- the way the scheme separates the checkpoint from the active data.

Figure 1 shows the taxonomy.

The sphere of BER is the part of the system that is checkpointed and, upon fault detection, rolls back to a previous state. Any scheme assumes that a datum that propagates outside the sphere is correct, because it has no mechanism to roll it back. Typically, the sphere of BER encloses the memory hierarchy up to a particular level. Consequently, we define three design points on this axis, depending on whether the sphere extends to the *registers*, the *caches*, or the *main memory* (Figure 2).

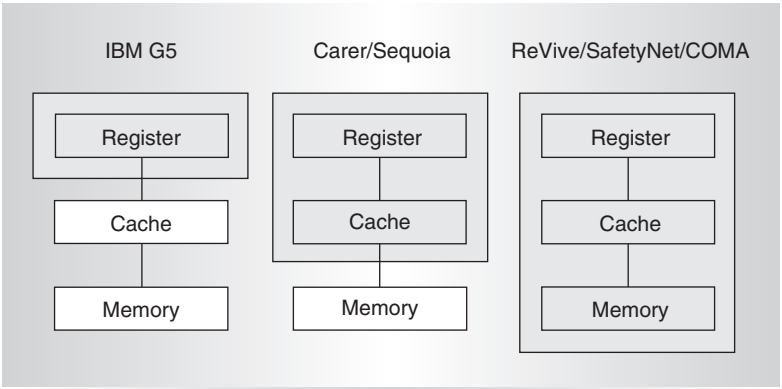

Figure 2. Design points based on the scope of the sphere of backward error recovery (BER).

The second axis in the taxonomy compares the level of the memory hierarchy checkpointed (M, where M is register, cache, or memory) and the level where the checkpoint is stored. This axis has two design points:[4] *Dual* means that the two levels are the same. This includes schemes that store the checkpoint in a special hardware structure closely attached to M. *Leveled* means the scheme saves the checkpoint at

Table 1. Characteristics of some existing hardware-based checkpointing and rollback schemes.

Scheme	Design point in the taxonomy	Hardware support	Tolerable fault detection latency	Recovery latency	Fault-free execution overhead
IBM G5	Register, dual, full	Redundant copy of register unit (R-unit), lock-stepping units	Pipeline depth	1,000 cycles	Negligible
Sparc64, out-of-order processor	Register, dual, renaming	Register alias table (RAT) copy and restore on branch speculation	Pipeline depth	10 to100 cycles	Negligible
Carer	Cache, dual, logging	Extra cache line state	Cache fill time	Cache invalidation	Not available
Sequoia	Cache, leveled, full	Cache flush	Cache fill time	Cache invalidation	Not available
Swich	Cache, dual, logging	Extra cache line states	Cache fill time	Cache invalidation	1%
ReVive	Memory, dual, logging	Memory logging, flush cache at checkpoint	100 ms	0.1 to 1 s	5%
SafetyNet	Memory, dual, logging	Cache and memory logging	0.1 to 1 ms	Not available	1%
Cache-Only Memory Architecture	Memory, dual, renaming	Mirroring by cache coherence protocol	2 to 200 ms	Not available	5 to 35%

a lower level than M—for example, a scheme might save a checkpoint of a cache in memory.

The third axis, which considers how the scheme separates the checkpoint from the active data, has two design points.[5] In *full separation,* the checkpoint is completely separated from the active data. In *partial separation,* the checkpoint and the active data are one and the same, except the data that has been modified since the last checkpoint. For this data, the checkpoint keeps both the old value at the time of the last checkpoint and the most current value.

Partial separation includes the subcategories *buffering, renaming,* and *logging.* In buffering, any modification to location L accumulates in a special buffer; at the next checkpoint, the modification is applied to L. In renaming, a modification to L does not overwrite the old value but is written to a different place, and the mapping of L is updated to point to this new place; at the next checkpoint, the new mapping of L is committed. In logging, a modification to L occurs in place, but the old value is copied elsewhere; at the next checkpoint, the old value is discarded.

Table 1 lists characteristics of several existing hardware-based checkpointing schemes and where they fit in our taxonomy. The "Checkpointing scheme comparison" sidebar provides more details on the schemes listed.

Trade-offs

The design points in our taxonomy in Figure 1 correspond to different trade-offs among fault detection latency, execution overhead, recovery latency, space overhead, type of faults supported, and hardware required. Figure 3 illustrates the trade-offs.

Consider the maximum tolerable fault detection latency. Moving from register-, to cache-, to memory-level checkpointing, the tolerable fault detection latency increases. This is because the corresponding checkpoints increase in size and can thus hold more execution history. Consequently, the time between when the error occurs and when it is detected can be larger while still allowing system recovery.

The fault-free execution overhead has an interesting shape. In register-level checkpointing, the overhead is negligible because these schemes save at most a few hundred bytes (for example, a few registers or the register alias table), and efficiently copy them nearby in hardware. At the other levels, a checkpoint has significant overhead, because the processors copy larger chunks of data over longer distances. An individual checkpoint's overhead tends to increase moving from cache- to memory-level checkpointing. However, designers tune the different designs' checkpointing frequencies to keep the overall

Checkpointing scheme comparison

Register-level checkpointing

IBM's S/390 G5 processor (in our taxonomy from Figure 1, *dual; full separation*) includes a redundant copy of the register file called the R-unit, which lets the processor roll back by one instruction.[1] In addition, it has duplicate lock-stepping instruction and execution units, and it can detect errors by comparing the signals from these units.

Fujitsu's Sparc64 processor leverages a mechanism present in most out-of-order processors: To handle branch misspeculation, processors can discard the uncommitted state in the pipeline and resume execution from an older instruction.[2] The register writes update renamed register locations (*dual; partial separation by renaming*). For this mechanism to provide recovery, it must detect faults before the affected instruction commits. In Sparc64, parity protects about 80 percent of the latches, and the execution units have a residue check mechanism—for instance, the Sparc64 checks integer multiplication using $\mathrm{mod}_3(A \times B) = \mathrm{mod}_3[\mathrm{mod}_3(A) \times \mathrm{mod}_3(B)]$.

Cache-level checkpointing

Carer checkpoints the cache by writing back dirty lines to memory (*leveled; full separation*).[3] An optimized protocol introduces a new cache line state called *unchangeable*. Under this protocol, at a checkpoint, the processor does not write back dirty lines; it simply write-protects them and marks them as unchangeable. Execution continues, and the processor lazily writes these lines back to memory either when they need to be updated or when space is needed (*dual; partial separation by logging*). For this optimization, the cache needs some forward error recovery (FER) protection such as error-correcting code (ECC), because the unchangeable lines in the cache are part of a checkpoint.

Sequoia flushes all dirty cache lines to the memory when the cache overflows or when an I/O operation is initiated.[4] Thus, the memory contains the checkpointed state (*leveled; full separation*).

Our proposed mechanism, Swich, is similar to Carer in that it can hold checkpointed data in the cache (*dual; partial separation by logging*). Unlike Carer, Swich holds two checkpoints simultaneously in the cache. Thus, Swich makes a large rollback window available continuously during execution.

Memory-level checkpointing

ReVive flushes dirty lines in the cache to memory at checkpoints.[5] It also uses a special directory controller to log memory updates in the memory (*dual; partial separation by logging*). SafetyNet logs updates to memory (and caches) in special checkpoint log buffers.[6] Therefore, we categorize SafetyNet's memory checkpointing as *dual; partial separation by logging.*

Morin et al. modified the cache coherence protocol of Cache-Only Memory Architecture (COMA) machines to ensure that at any time, every memory line has exactly two copies of data from the last checkpoint in two distinct nodes.[7] Thus, the machines can recover the memory from single-node failures (*dual; partial separation by renaming*).

References

1. T.J. Slegel et al., "IBM's S/390 G5 Microprocessor Design," *IEEE Micro,* vol. 19, no. 2, Mar.-Apr. 1999, pp. 12-23.
2. H. Ando et al., "A 1.3GHz Fifth Generation SPARC64 Microprocessor," *IEEE J. Solid-State Circuits,* vol. 38, no. 11, Nov. 2003, pp. 1896-1905.
3. D.B. Hunt and P.N. Marinos, "General-Purpose Cache-Aided Rollback Error Recovery (CARER) Technique," *Proc. 17th Int'l Symp. Fault-Tolerant Computing Systems* (FTCS 87), IEEE CS Press, 1987, pp. 170-175.
4. P.A. Bernstein, "Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing," *Computer,* vol. 21, no. 2, Feb. 1988, pp. 37-45.
5. M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors," *Proc. 29th Ann. Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 111-122.
6. D.J. Sorin et al., "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery," *Proc. 29th Ann. Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 123-134.
7. C. Morin et al., "COMA: An Opportunity for Building Fault-Tolerant Scalable Shared Memory Multiprocessors," *Proc. 23rd Ann. Int'l Symp. Computer Architecture* (ISCA 96), IEEE CS Press, 1996, pp. 56-65.

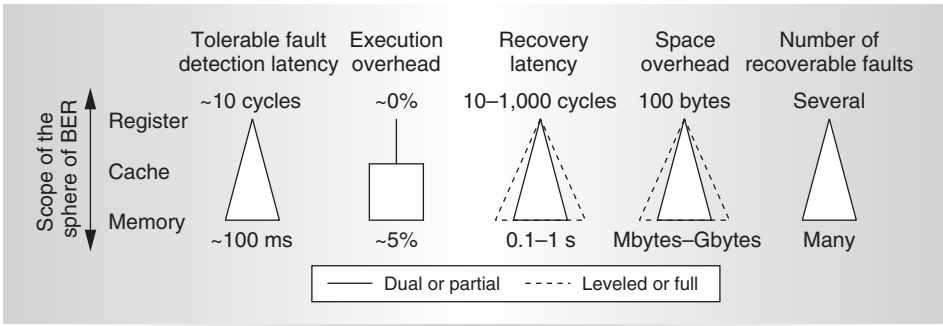

Figure 3. Trade-offs for design points in the checkpointing taxonomy.

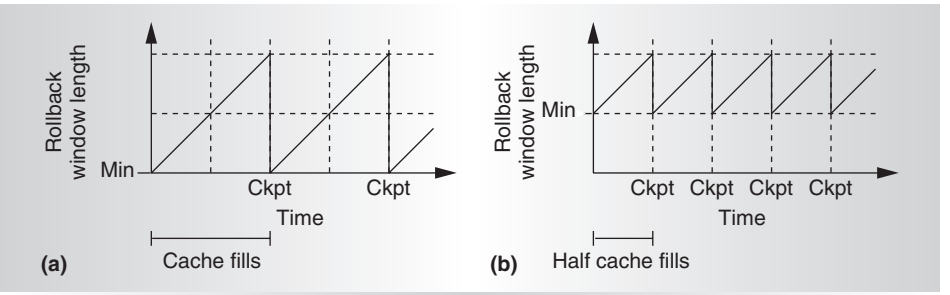

Figure 4. Evolution of the rollback window length with time for Sequoia and Carer (a) and for Swich (b).

execution time overhead tolerable—about 5 percent, as shown in Figure 3.

The designs with relatively more expensive individual checkpoints take less frequent checkpoints. As a result, both their recovery latency and their space overhead increase along the axis from cache- to memory-level checkpoints. The trend is the same moving from dual systems (solid lines in Figure 3) to leveled systems (dashed lines) because individual checkpoints in leveled systems involve moving data between levels and, therefore, are more expensive. A similar effect also occurs as we move from partial- (solid lines in Figure 3) to full-separation designs (dashed lines), which require more data copying.

Finally, moving from register-, to cache-, to memory-level checkpointing, the number of recoverable faults increases. The reason is twofold: First, these schemes support more faults of a given type, thanks to the longer detection latencies tolerated. Second, they support more types of faults (for example, faults that occur at the new hierarchy level). However, recovering from these faults requires additional hardware support, which is more expensive because it is spread more widely throughout the machine.

Motivation for the Swich architecture

Clearly, the different checkpointing schemes address a large set of trade-offs; our main focus in this article is cache-level checkpointing schemes with their advantages and drawbacks.

On the positive side, trends in technology are making the cache level more attractive. More on-chip integration means the possibility of larger caches, which let cache-level checkpointing offer increased fault detection latencies. In addition, the hardware for cache-level checkpointing is relatively inexpensive and functions efficiently in modern architectures. Specifically, the primary hardware these schemes require is new cache line states to mark lines related to the checkpoint. Modification of the cache replacement algorithm might be necessary to encourage these lines to remain cached. Cache-level checkpointing schemes also need to change the state of groups of cache lines at commit and rollback points. Schemes can support this by accessing the cache tags one at a time to change each concerned tag individually, or by adding hardware signals that modify all the concerned tags in one shot. Finally, processors using cache-level checkpointing must save registers efficiently at a checkpoint and restore them at a rollback—similarly to the way processors handle branch speculation.

On the negative side, it is difficult to control the checkpointing frequency in cache-level checkpointing schemes because the displacement of a dirty cache line triggers a checkpoint.[6] Such displacements are highly dependent on the application and the cache organization. Thus, a system's performance with a cache-level checkpointing scheme is relatively unpredictable.

Another problem of existing implementations of cache-level checkpointing[2,3] is that the length of the code that they can roll back (the rollback window) is at times very small. Specifically, suppose a fault occurs immediately before a checkpoint, and is only detected after the checkpoint has completed (when the data from the previous checkpoint has been discarded). At this point, the amount of code that can be rolled back is very small, or even zero. Figure 4a shows how the rollback

window length evolves in existing cache-level checkpointing schemes. Right after each checkpoint, the window empties completely.

Sliding-window cache-level checkpointing

Swich addresses the problem of the empty rollback window by providing a *sliding* rollback window, increasing the chances that the rollback window always has a significant minimum length. The basic idea is to always keep in the cache the state generated by two uncommitted checkpoint intervals, which we call *speculative epochs*. Each epoch lasts as long as it takes to generate a dirty data footprint about half the cache size. When the cache is getting full of speculative data, the older epoch is committed and a newer one starts. With this approach, the evolution of the rollback window length is closer to that illustrated in Figure 4b. When rollback is necessary, the scheme rolls back both speculative epochs. In the worst case (when rollback is necessary immediately after a checkpoint), the system can roll back at least one whole speculative epoch. Previously proposed schemes have a worst-case scenario of an empty rollback window.

Checkpoint, commit, and rollback

We extend each line in the cache with two epoch bits, E_1E_0. If the line contains nonspeculative (that is, committed) data, E_1E_0 are 00; if it contains data generated by one of the two active speculative epochs, E_1E_0 are 01 or 10. Only dirty lines (those with the Dirty bit set) can have E_1E_0 other than 00.

A checkpoint consists of three steps: committing the older speculative epoch, saving the current register file and processor state, and starting a new speculative epoch with the same E_1E_0 identifier as the one just committed. Committing a speculative epoch involves discarding its saved register file and processor state and clearing the epoch bits of all the cache lines whose E_1E_0 bits are equal to the epoch's identifier. It is preferable to clear such bits all in one shot, with a hardware signal connected to all the tags that takes no more than a handful of cycles to actuate.

On a fault, the two speculative epochs roll back. This involves restoring the older of the two saved register files and processor states, invalidating all the cache lines whose E_1E_0 bits are not 00, and clearing all the E_1E_0 bits. A one-shot hardware signal is sufficient for the operations on the E_1E_0 bits. However, because rollback is probably infrequent, these operations can also use slower hardware that walks the cache tags and individually sets the bits of the relevant tags. For a write-through L1 cache, it is simpler to invalidate the entire cache.

Speculative line management

Swich manages the speculative versions of lines, namely those belonging to the speculative epochs, following two invariants. First, only writes can create speculative versions of lines. Second, the cache can contain only a single version of a given line. Such a version can be nonspeculative or speculative with a given E_1E_0 identifier. In the latter case, the line is necessarily dirty and cannot be displaced from the cache. Following the first invariant, a processor read simply returns the data currently in the cache and does not change its line's E_1E_0 identifier. If the read misses in the cache, the line is brought from memory, and E_1E_0 are set to 00.

The second invariant determines the actions on a write. There are four cases: First, if the write misses, the line is brought from memory, it is updated in the cache, and its E_1E_0 bits are set to the epoch identifier. Second, if the write hits on a nonspeculative line in the cache, the line is first written back to memory (if dirty), then it is updated in the cache, and finally its E_1E_0 bits are set to the epoch identifier. In these two cases, a count of the number of lines belonging to this epoch (SpecCnt) is incremented. Third, if the write hits on a speculative line of the same epoch, the update proceeds normally. Finally, if the write from current epoch i hits on a speculative line from the other epoch ($i - 1$), then epoch $i - 1$ is committed, and the write initiates a new epoch, $i + 1$. The line is written back to memory, then updated in the cache, and finally its E_1E_0 bits are set to the new epoch identifier. When the SpecCnt count of a given epoch (say, epoch i) reaches the equivalent of half the cache size, epoch $i - 1$ is committed, and epoch $i + 1$ begins.

Recall that speculative lines cannot be displaced from the cache. Consequently, if space is needed in a cache set, the algorithm first chooses a nonspeculative line as its victim. If all lines in the set are speculative, Swich commits

the older speculative epoch (say, epoch $i-1$) and starts a new epoch (epoch $i+1$). Then, the algorithm chooses one of the lines just committed as a victim for displacement.

A fourth condition that also leads to an epoch commitment occurs when the current epoch (say epoch i) is about to use all the lines of a given cache set. If the scheme allows it to do so, the sliding window becomes vulnerable: If the epoch later needs an additional line in the same set, the hardware would have to commit i (and $i-1$), decreasing the sliding window size to zero. To avoid this, when an epoch is about to use all the lines in a set, the hardware first commits the previous epoch ($i-1$) and then starts a new epoch with the access that requires a new line.

Finally, in the current design, when the program performs I/O or issues a system call that has side effects beyond generating cached state, the two speculative epochs are committed. At that point, the size of the sliding window falls to zero.

Two-level cache hierarchies

In a two-level cache hierarchy, the algorithm we have described is implemented in the L2 cache. The L1 cache can use a simpler algorithm and does not need the E_1E_0 bits per line. For example, consider a write-through, no-allocate L1. On a read, L1 provides the data that it has or that it obtains from L2. On a write hit, L1 is updated, and both the update and epoch bits are propagated to L2. On a commit, no action is taken, whereas in the rare case of a rollback, all of L1 is invalidated.

Multiprocessor issues

Swich is compatible with existing techniques that support cache-level checkpointing in a shared-memory multiprocessor.[7,8] For example, assume we use the approach of Wu et al.[8] In this case, when a cache must supply a speculative dirty line to another cache, the source cache must commit the epoch that owns the line to ensure that the line's data never rolls back to a previous value. The additional advantage of Swich is that the source cache needs only to commit a single speculative epoch, if the line provided belonged to its older speculative epoch. In this case, the rollback window length in the source cache does not fall to zero. A similar operation occurs when a cache receives an invalidation for a speculative dirty line. The epoch commits, and the line is provided and then invalidated. Invalidations to all other types of lines proceed as usual. A similar discussion can be presented for other existing techniques for shared-memory multiprocessor checkpointing.

Swich prototype

We implemented a hardware prototype of Swich using FPGAs. As the base for our implementation, we used Leon2, a 32-bit processor core compliant with the Sparc V8 architecture (http://www.gaisler.com). Leon2 is in synthesizable VHDL format. It has an in-order, single-issue, five-stage pipeline. Most instructions take five cycles to complete if no stalls occur. The processor has a windowed register file, and the instruction cache is 4 Kbytes. The data cache size, associativity, and line size are configurable. In our experiments, we set the line size to 32 bytes and the associativity to 8, and we varied the cache size. Because the processor initially had a write-through data cache, we implemented a write-back data cache controller.

We extended the processor in two major ways: with cache support for buffering speculative data and rollback, and with register support for checkpointing and rollback. We also leveraged, in part, the rollback mechanism presented elsewhere.[9]

Data cache extensions

Although the prototype implements most of the design we described earlier, there are a few differences. First, the prototype has a single-level data cache. Second, for simplicity, we allowed a speculative epoch to own at most half the lines in any cache set. Third, the prototype doesn't perform epoch commit and rollback using a one-shot hardware signal. Instead, we designed a hardware state machine that walks the cache tags performing the operations on the relevant cache lines. The reason for this design is that we implemented the cache with the synchronous RAM blocks present in the Xilinx Virtex II series of FPGAs. Such memory structures have only the ordinary address and data inputs; they are difficult to modify to incorporate additional control signals, such as those needed to support one-shot commit and rollback signals.

We extended the cache controller with a cache walk state machine (CWSM) that traverses the cache tags and clears the corresponding epoch bits (in a commit) and valid bits (in a rollback). A commit or rollback signal activates the CWSM. At that point, the pipeline stalls and the CWSM walks the cache tags. The CWSM has three states, shown in Figure 5a: *idle*, *walk*, and *restore*. In *idle*, the CWSM is inactive. In *walk*, its main working state, the CWSM can operate on one or both of the speculative epochs. The traversal takes one cycle for each line in the cache. The *restore* state resumes the normal cache controller operation and releases the pipeline.

When operating system code is executed, we conservatively assume that it cannot be rolled back and, therefore, we commit both speculative epochs and execute the system code nonspeculatively. A more careful implementation would identify the sections of system code that have no side effects beyond memory updates, and allow their execution to remain speculative.

Register extensions

Swich performs register checkpointing and restoration using two shadow register files, SRF0 and SRF1. Each of these memory structures is identical to the main register file. When a checkpoint is to be taken, the pipeline stalls and passes control to the register checkpointing state machine (RCSM), which has the four states shown in Figure 5b.

The RCSM is in the *idle* state while the pipeline executes normally. When a register checkpoint must be taken, it transitions to the *checkpoint* state. In this state, the RCSM copies the valid registers in the main register file to one of the SRFs. The register files are implemented in SRAM and have two read ports and one write port. This means that the RCSM can copy only one register per cycle. Thus, the *checkpoint* stage takes about as many cycles as there are valid registers in the register file. The *rollback* state is activated when the pipeline receives a rollback signal. In this state, the RCSM restores the contents of the register file from the checkpoint. Similarly, this takes about as many cycles as there are valid registers. The *restore* state reinitializes the pipeline.

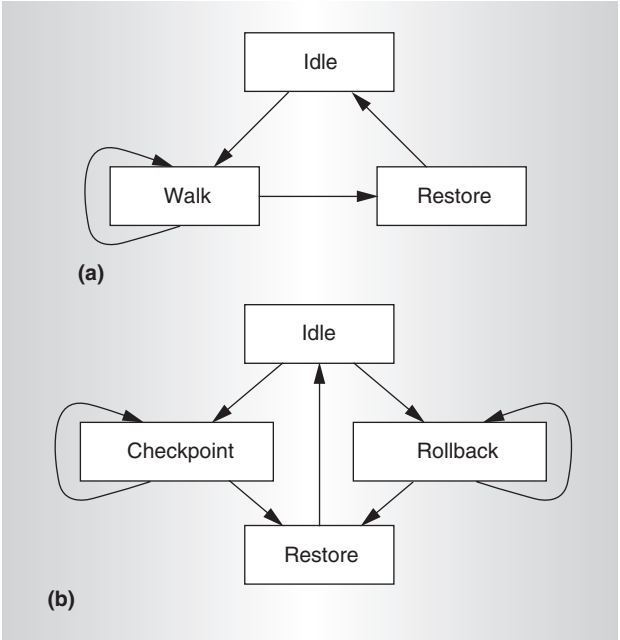

Figure 5. Swich state machines: cache walk state machine (a) and register checkpointing state machine (b).

Evaluating the Swich prototype

The processor is part of a SoC infrastructure that includes a synthesizable SDRAM controller, and PCI and Ethernet interfaces. We synthesized the system using Xilinx ISE v6.1.03 (http://www.xilinx.com). The target FPGA chip was a Xilinx Virtex II XC2V3000 running on a GR-PCI-XC2V development board (http://www.pender.ch). The processor runs at 40 MHz, and the board has 64 Mbytes of SDRAM for main memory. Communication with the board and loading of programs in memory take place through the PCI interface from a host computer.

On this hardware, we run a special version of the SnapGear Embedded Linux distribution (http://www.snapgear.org). SnapGear Linux is a full-source package, containing kernel, libraries, and application code for rapid development of embedded Linux systems. We used a cross-compilation tool chain for the Sparc architecture to compile the kernel and applications.

We experimented with the prototype using a set of applications that included open-source Linux applications and microbenchmarks that we ran atop SnapGear Linux. We chose codes that exhibited a variety of memory and system code access patterns to give us a sense of

Table 2. Applications evaluated using Swich.

Application	Description
hennessy	Collection of small, computation-intensive applications; little or no system code
polymorph	Converts Windows-style file names to Unix; moderate system code
memtest	Microbenchmark that simulates large array traversals; no system code
sort	Linux utility that sorts lines in text files; system-code intensive
ncompress	Compression and decompression utility; moderate system code
ps	Linux process status command; very system-code intensive

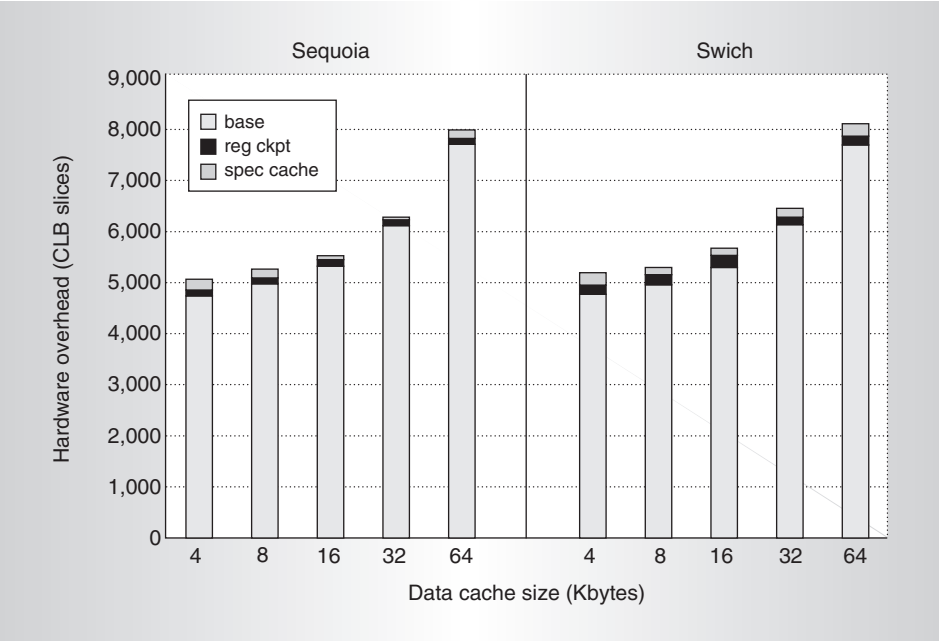

Figure 6. Amount of logic consumed by different designs and components.

how workload characteristics affect Swich. Table 2 describes the applications we used.

Results

We have not yet accurately measured Swich's execution overhead or recovery latency, because of the limitations of our implementation—specifically that we implemented epoch commit and rollback with inefficient cache tag walking rather than with an efficient one-shot hardware signal. This problem slows down the Swich FPGA prototype, but it would not be present in a CMOS implementation.

Consequently, we focused our evaluation on measuring Swich's hardware overhead and rollback window size, and on roughly estimating Swich's performance overhead and recovery latency using a simple model.

Hardware overhead

For each of Swich's two components—the register checkpointing mechanism and the data cache support for speculative data—we measured the hardware overhead in both logic and memory structures. Our measurements considered only the processor core and caches, not the other on-chip modules such as the PCI and memory controllers. As a reference, we also measured the overheads of a cache checkpointing scheme with one speculative epoch at a time (a system similar to Sequoia[3]).

Figure 6 shows the logic consumed, measured in number of configurable logic block (CLB) slices. CLBs are the FPGA blocks used to build combinational and synchronous logic components. A CLB comprises four similar slices; each slice includes two four-input function generators, carry logic, arithmetic logic

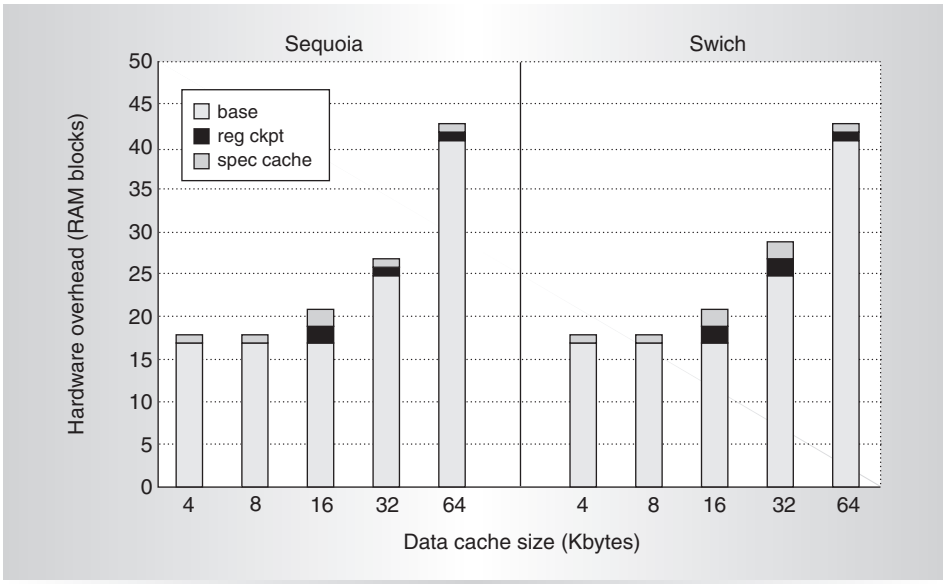

Figure 7. Amount of RAM consumed by the different designs and components.

gates, wide-function multiplexers, and two storage elements. Figure 6 graphs hardware overhead for the Sequoia-like system and the Swich system, each with several different data cache sizes (ranging from 4 Kbytes to 64 Kbytes). Each bar shows the CLBs consumed by the logic to support speculative data in the cache (spec cache), register checkpointing (reg ckpt), and the original processor (base).

With Swich, the logic overhead of the combined extensions was only 6.5 percent on average and relatively constant across the range of caches we evaluated; the support we propose does not use much logic. In addition, the number of CLBs used for Swich is only about 2 percent higher than for the Sequoia-like system. In other words, the difference in logic between a simple window and a sliding window is small.

Figure 7 shows the memory consumed, measured in number of SelectRAM memory blocks. These blocks, used to implement the caches, register files, and other structures, are 18-Kbit, dual-port RAM blocks with two independently clocked and independently controlled synchronous ports that access a common storage area. The overhead in memory blocks of the combined Swich extensions is again small for all the caches that we evaluated, indicating that the support we propose consumes few hardware resources. In addition, Swich's overhead in memory blocks is again very similar to that of the Sequoia-like system. The Swich additions are primarily additional register checkpoint storage and additional state bits in the cache tags.

Size of the rollback window

We particularly wanted to determine the size of the Swich rollback window in number of dynamic instructions. This number determines how far the system can roll back at a given time if it detects a fault. We were especially interested in the minimum points in Figure 4b, which represent the cases when only the shortest rollback windows are available. Figure 4b shows an ideal case. In practice, the minima aren't all at the same value. The actual values of the minimum points depend on the code being executed: the actual memory footprint of the code and the presence of system code. In our current implementation, execution of system code causes the commit of both speculative epochs and, therefore, induces a minimum in the rollback window.

For our experiments, we measured the window size at each of the minima and took per-application averages. Figure 8 shows the average window size at minimum points for the different applications and cache sizes ranging from 8 Kbytes to 64 Kbytes. The same information for the Sequoia and Carer systems would show rollback windows of size zero.

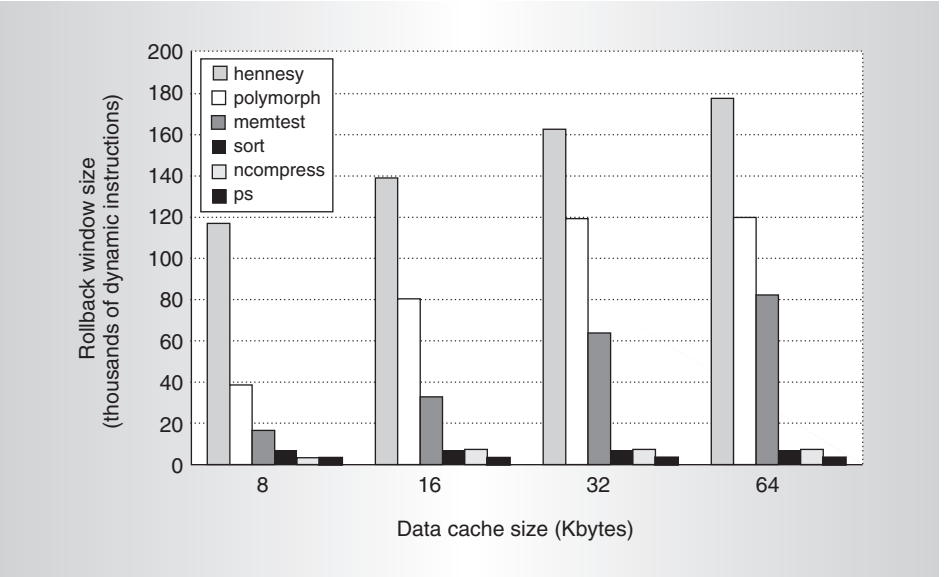

Figure 8. Minimum rollback window size in Swich, per-application averages.

As Figure 8 shows, the hennessy microbenchmark has the largest minimum rollback window. The application is computationally intensive, has little system code, and has a relatively small memory footprint. As the cache increases, the window size increases. For 8- to 64-Kbyte caches, the minimum rollback window ranges from 120,000 to 180,000 instructions. The polymorph and memtest applications have more system code and a larger memory footprint, respectively. Their rollback windows are smaller, although they increase substantially with the cache size, to reach 120,000 and 80,000 instructions, respectively, for the 64-Kbyte cache. The sort, ncompress, and ps applications are system-code-intensive applications with short rollback windows, regardless of the cache size. One way to increase their minimum rollback windows is to identify the sections of system code that have no side effects beyond memory updates and allow their execution to remain speculative.

For completeness, we also measured Swich's average rollback window size and compared it to that of the implementation similar to Sequoia and Carer. Again, we looked at four different data cache sizes and six applications; Figure 9 shows the results. Except in system-code-intensive applications, Swich's average rollback window is significantly larger than that of Sequoia-Carer. The reason, as Figure 8 shows, is that for Swich the rollback window has a nonzero average minimum value. For ps and the other two system-intensive applications, there is little difference between the two schemes. Overall, at least for applications without frequent system code, Swich supports a large minimum rollback window.

Estimating performance overhead and recovery latency

A rough way of estimating Swich's performance overhead is to compute the ratio between the time needed to generate a checkpoint and the duration of the execution between checkpoints. This approach, of course, neglects any overheads that the scheme might induce between checkpoints. To model worst-case conditions, we used the average size of the rollback window at minimum points (Figure 8) as the number of instructions between checkpoints. Across all the applications, this is about 60,000 instructions for 64-Kbyte caches. If we assume an average of one cycle per instruction, this corresponds to 60,000 cycles. On the other hand, a checkpoint can very conservatively take about 256 cycles, because it involves saving registers and asserting a one-shot hardware signal that modifies cache tag bits. Overall, the ratio results in a 0.4 percent overhead.

We estimated the recovery latency by assuming 256 cycles to both restore the registers and

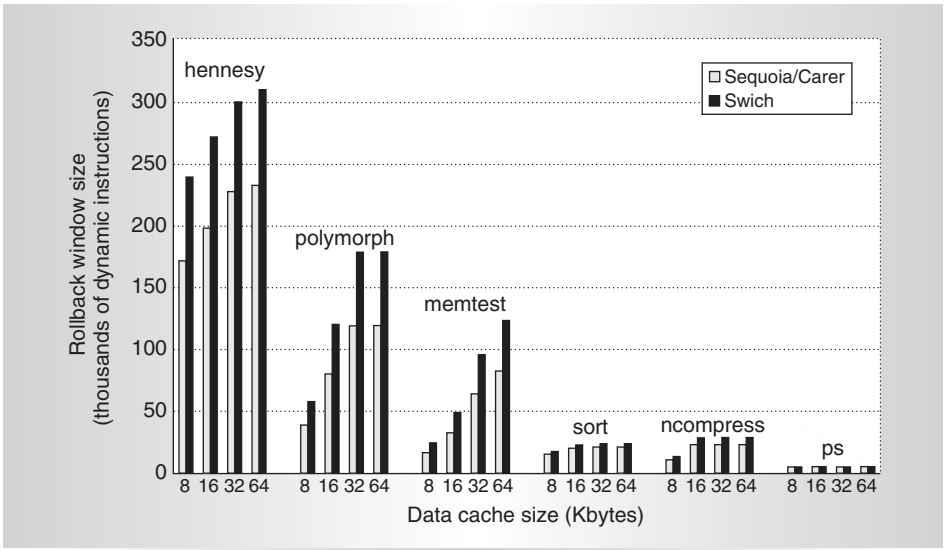

Figure 9. Comparison of average rollback window size for an implementation similar to Sequoia and Carer and for Swich.

assert a one-shot hardware signal that modifies cache tag bits. In addition, as the processor reexecutes, it must reload into the cache all the dirty speculative lines that were invalidated during rollback. Overall, the combined overhead of both effects is likely to be in the milliseconds at most. Given the low frequency of rollbacks, we believe this is tolerable.

Cache-level checkpointing provides an efficient recovery mechanism that can be implemented inexpensively in modern processors. This work shows that maintaining two speculative epochs at all times to form a sliding rollback window lets Swich support a large minimum (and average) rollback window. Moreover, our FPGA prototype showed that the added hardware is minimal and that it can be well integrated with existing processor microarchitectures. We expect that supporting Swich will significantly help processors recover from transient and intermittent faults, as well as from software errors that have low detection latency. We are currently examining the behavior of Swich under a wider range of workloads. We are also examining how Swich can be supported while the processor executes system code. MICRO

Acknowledgments

Partial support for this work came from the National Science Foundation under grants EIA-0072102, EIA-0103610, CHE-0121357, and CCR-0325603; from DARPA under grant NBCH30390004; from the US Department of Energy under grant B347886; and from IBM and Intel.

References

1. D.P. Siewiorek and R.S. Swarz, *Reliable Computer Systems: Design and Evaluation,* 3rd ed., AK Peters, 1998.
2. D.B. Hunt and P.N. Marinos, "General-Purpose Cache-Aided Rollback Error Recovery (CARER) Technique," *Proc. 17th Int'l Symp. Fault-Tolerant Computing Systems* (FTCS 87), IEEE CS Press, 1987, pp. 170-175.
3. P.A. Bernstein, "Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing," *Computer,* vol. 21, no. 2, Feb. 1988, pp. 37-45.
4. N.S. Bowen and D.K. Pradhan, "Processor- and Memory-Based Checkpoint and Rollback Recovery," *Computer,* vol. 26, no. 2, Feb. 1993, pp. 22-31.
5. M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors," *Proc. 29th Ann. Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 111-122.
6. B. Janssens and W.K. Fuchs, "The Performance of Cache-Based Error Recovery in Multiprocessors," *IEEE Trans. Parallel and*

Distributed Systems, vol. 5, no. 10, Oct. 1994, pp. 1033-1043.
7. R.E. Ahmed, R.C. Frazier, and P.N. Marinos, "Cache-Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multiprocessor Systems," *Proc. 20th Int'l Symp. Fault-Tolerant Computing* (FTCS 90), IEEE CS Press, 1990, pp. 82-88.
8. K.L. Wu, W.K. Fuchs, and J.H. Patel, "Error Recovery in Shared-Memory Multiprocessors Using Private Cache," *IEEE Trans. Parallel and Distributed Systems,* vol. 1, no. 10, Apr. 1990, pp. 231-240.
9. R. Teodorescu and J. Torrellas, "Prototyping Architectural Support for Program Rollback Using FPGAs," *IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM), IEEE CS Press, 2005, pp. 23-32.

Radu Teodorescu is a PhD candidate in the Computer Science Department of the University of Illinois at Urbana-Champaign. His research interests include processor design for reliability, debuggability, and variability tolerance. Teodorescu has a BS in computer science from the Technical University of Cluj-Napoca, Romania, and an MS in computer science from University of Illinois at Urbana-Champaign. He is a student member of the IEEE.

Jun Nakano is an Advisory IT Specialist at IBM Japan. His research interests include reliability in computer architecture and variability in semiconductor manufacturing. Nakano has an MS in physics from the University of Tokyo and a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of the IEEE and the ACM.

Josep Torrellas is a professor of computer science and Willett Faculty Scholar at the University of Illinois at Urbana-Champaign. He also chairs the IEEE Technical Committee on Computer Architecture (TCCA). His research interests include multiprocessor computer architecture, thread-level speculation, low-power design, reliability, and debuggability. Torrellas has a PhD in electrical engineering from Stanford University. He is an IEEE Fellow and a member of the ACM.

Direct questions and comments about this article to Radu Teodorescu, Siebel Center for Computer Science, Room 4238, 201 North Goodwin Ave., Urbana, IL 61801; teodores@cs.uiuc.edu.