SWICH: A Prototype for Efficient Cache-Level Checkpointing and Rollback *

Radu Teodorescu, **Jun Nakano**, and **Josep Torrellas**
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Abstract

Low-overhead checkpointing and rollback is a popular technique for fault recovery. While different approaches are possible, hardware-supported checkpointing and rollback at the cache level is especially interesting. The main reason is that the microarchitecture required can be easily integrated and used effectively in modern processor microarchitectures. Unfortunately, proposed cache-level checkpointing schemes provide little implementation information and do not support a large rollback window all the time.

In this paper, we outline the design of SWICH, a new cache-level checkpointing scheme that, while being efficiently implemented in modern processors, supports a large rollback window *continuously*. This is accomplished by relying on two live checkpoints at all times to form a *sliding* rollback window. We build an FPGA-based prototype of SWICH. The evaluation of the prototype shows that a simple processor can support the sliding window with little additional logic and memory hardware. Moreover, for applications without frequent I/O or system calls, the technique sustains large minimum and average rollback windows.

Keywords: *transient faults, low-overhead checkpointing, hardware prototype.*

1. Introduction

Backward recovery through checkpointing and rollback is a popular approach to recover from transient and intermittent faults [12]. This approach is especially attractive when implemented with very low overhead, typically thanks to dedicated hardware support.

There are multiple proposals for low-overhead checkpointing and rollback schemes (e.g., [13, 7, 3, 11, 14, 9]). These proposals differ in many ways, including what level of the memory hierarchy is checkpointed and how the checkpoint is organized relative to the active data. They correspond to different trade-offs between tolerable fault detection latency, execution and space overhead, recovery latency, type of faults supported, and hardware required.

An especially interesting design point is schemes that checkpoint at the cache level [3, 7]. In these schemes, the data in the cache is regularly saved in a checkpoint, which can later be used for fault recovery. What makes cache-level schemes interesting is that the hardware required can be integrated relatively inexpensively in modern processor microarchitectures and used efficiently. Moreover, technology trends are allowing more on-chip integration, which in turn enables larger caches. The result is an increase in the fault detection latency of cache-level checkpointing schemes.

Unfortunately, proposed cache-level checkpointing schemes such as CARER [7] and Sequoia [3] do not support a large rollback window all the time. The rollback window at a given time is the set of instructions that can be "undone" if a fault is detected by returning to a previous checkpoint. Specifically, in these schemes, immediately after a checkpoint is taken, the window of code that can be rolled back is typically reduced to nil.

In this paper, we describe the microarchitecture of a new cache-level checkpointing scheme that, while being efficiently and inexpensively implemented in modern processors, supports a large rollback window *at all times*. This is accomplished by keeping in the cache data from two consecutive checkpoints (or epochs) at all times. When the older epoch needs to commit to make room for data in the cache, a new epoch starts. With this approach, the rollback window typically decreases by half when one epoch commits, rather than to zero, and then slowly increases until the next commit. We refer to this approach as a sliding rollback window, and to our scheme as *SWICH*, for Sliding Window Cache-Level Checkpointing.

Moreover, we implement a prototype of SWICH using Field-Programmable Gate Arrays (FPGAs). The evaluation of the prototype shows that supporting cache-level checkpointing with sliding window is promising. Specifically, this technique adds little additional logic and memory hardware to a simple processor. Moreover, for applications without frequent I/O or system calls, the technique sustains

*This work was supported in part by the National Science Foundation under grants EIA-0072102, EIA-0103610, CHE-0121357, and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel.

a large minimum rollback window.

Overall, this paper makes three contributions. First, we present a taxonomy and trade-off analysis of low-overhead checkpointing schemes. Secondly, we present the microarchitecture of SWICH. Finally, we implement a SWICH prototype using FPGAs and evaluate it.

The paper is organized as follows: Section 2 presents a taxonomy and trade-off analysis of low-overhead checkpointing schemes; Section 3 presents the microarchitecture of SWICH; Section 4 discusses the prototype; Section 5 evaluates the prototype; and Section 6 concludes.

2 Low-Overhead Checkpointing Schemes

2.1 A Taxonomy

To help in our presentation of SWICH, we find it useful to classify hardware-based checkpointing and rollback schemes into a taxonomy with three axes: (i) the scope of the sphere of Backward Error Recovery (BER), (ii) the relative location of the checkpoint and active data, and (iii) how the checkpoint is separated from the active data. Active data are the most current versions of the data. Figure 1 shows the taxonomy.

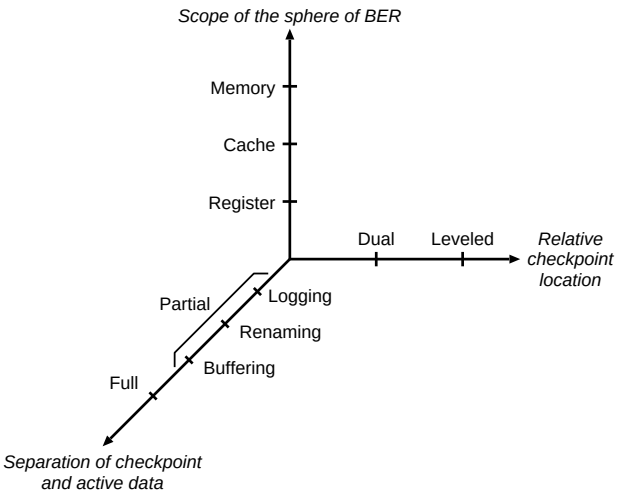

Figure 1. A taxonomy of hardware-based checkpointing and rollback schemes.

The sphere of BER is the part of the system that is checkpointed and, in case of a fault, can be rolled back to a previous state. Any datum that propagates outside of the sphere is assumed to be correct, since there is no mechanism to roll it back. Typically, the sphere of BER encloses up to a given level of the memory hierarchy. Consequently, we define 3 design points, namely sphere that encloses the registers, the caches, and the main memory (Figure 2).

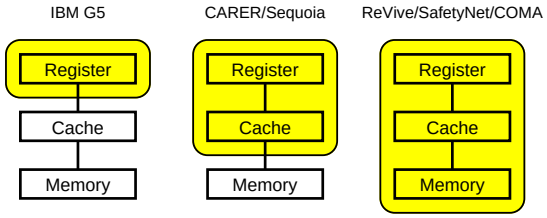

Figure 2. Design points based on the scope of the sphere of Backward Error Recovery (BER).

The second axis compares the level of the memory hierarchy that is checkpointed ($\mathcal{M}$, where $\mathcal{M}$ is register, cache, or memory), and the level where the checkpoint is stored. There are two design points [4]. The first one is *dual*, where both levels are the same. This includes the case when the checkpoint is stored in some special hardware structure that is closely attached to $\mathcal{M}$. The second design is *leveled*, where the checkpoint is saved in a lower level than $\mathcal{M}$ (e.g., the checkpoint of a cache is saved in memory).

The third axis considers how the checkpoint is separated from the active data. It has two design points [11]. 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 for the data that have been modified since the last checkpoint. For these data, we need to keep both the old value at the time of the last checkpoint and the most current value. Partial separation is further categorized into buffering, renaming, and logging. In *buffering*, any modification to location $\mathcal{L}$ is accumulated in a special buffer; at the next checkpoint, the modification is applied to $\mathcal{L}$. In *renaming*, a modification to $\mathcal{L}$ does not overwrite the old value but is written to a different place, and the mapping of $\mathcal{L}$ is updated to point to this new place; at the next checkpoint, the new mapping of $\mathcal{L}$ is committed. In *logging*, a modification to $\mathcal{L}$ occurs in place, but the old value is copied elsewhere; at the next checkpoint, the old value is discarded.

2.2 Examples

Table 1 lists the characteristics of some of the existing hardware-based checkpointing schemes and where they fit in our taxonomy. We describe them next.

2.2.1 Register-Level Checkpointing

The IBM's S/390 G5 processor [13] has a redundant copy of the register file called *R-unit* (*dual; full separation*), which enables rolling back by one instruction. In addition, it has duplicate lock-stepping instruction and execution units, and

Scheme	Design Point in the Taxonomy	Hardware Support	Tolerable Fault Detection Latency	Recovery Latency	Fault-free Execution Overhead
IBM G5	Register/Dual/Full	R-unit file, lock-stepping units	Pipeline depth	$\sim$ 1000 cycles	Negligible
OOO Processor	Register/Dual/Renaming	RAT copy/restore on branch speculation	Pipeline depth	10–100 cycles	Negligible
CARER	Cache/Dual/Logging	Extra cache line state	$\sim$ cache fill time	Cache invalidation	Not available
Sequoia	Cache/Leveled/Full	Cache flush	$\sim$ cache fill time	Cache invalidation	Not available
SWICH	Cache/Dual/Logging	Extra cache line states	$\sim$ cache fill time	Cache invalidation	$<$ 1%
ReVive	Memory/Dual/Logging	Memory logging, flush cache at checkpoint	$\sim$ 100 msec	0.1–1 sec	$\sim 5\%$
SafetyNet	Memory/Dual/Logging	Cache/memory logging	0.1–1 msec	Not available	$\sim 1\%$
COMA	Memory/Dual/Renaming	Mirroring by cache coherence protocol	2–200 msec	Not available	5–35%

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

can detect errors by comparing the signals from these units.

Fujitsu's SPARC64 processor [2] leverages a mechanism that is 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. Register writes update renamed register locations (*dual; partial separation by renaming*). For this mechanism to provide recovery, faults must be detected before the affected instruction commits. In SPARC64, about 80% of the latches are protected by parity and the execution units have a residue check mechanism — for instance, integer multiplication is checked using $mod_3(A \times B) = mod_3(mod_3(A) \times mod_3(B))$.

2.2.2 Cache-Level Checkpointing

CARER [7] checkpoints the cache by writing back dirty lines to memory (*leveled; full separation*). An optimized protocol introduces a new cache line state called *unchangeable*. In this case, at a checkpoint, dirty lines are not written back; they are simply write-protected and marked as unchangeable. Execution continues and these lines are written back to memory lazily when they need to be updated or space is needed (*dual; partial separation by logging*). Note that, for this optimization, the cache needs some FER (Forward Error Recovery) protection such as ECC, since the unchangeable lines in the cache are part of a checkpoint.

Sequoia [3] flushes all dirty cache lines to the memory when the cache overflows or an I/O operation is initiated. Thus, the memory contains the checkpointed state (*leveled; full separation*).

Our proposed mechanism (*SWICH*, which is described in Section 3), 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. This allows it to provide a *larger* rollback window during execution.

2.2.3 Memory-Level Checkpointing

ReVive [11] flushes dirty lines in the cache to memory at checkpoints. From then on and until the next checkpoint, it uses a special directory controller to log memory updates in the memory (*dual; partial separation by logging*).

In SafetyNet [14], updates to memory (and caches) are logged in special checkpoint log buffers. Therefore, memory checkpointing is categorized as *dual; partial separation by logging*.

Morin *et al.* [9] modified the cache coherence protocol of COMA (Cache Only Memory Architecture) machines. The new protocol ensures that at any time, every memory line has exactly two copies of data from the last checkpoint in two distinct nodes so that the memory can be recovered from one node failure (*dual; partial separation by renaming*).

2.3 Trade-offs

The different design points in our taxonomy of Figure 1 correspond to different trade-offs in 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. As we move from register to cache and to memory-level checkpointing, the tolerable fault detection latency of the schemes increases. This is because the size of the corresponding memory hierarchy level increases. Consequently, a faulty datum takes longer to propagate outside the sphere of BER.

The fault-free execution overhead has an interesting shape. In register-level checkpointing, the overhead is negligible because only a few hundred bytes or less are copied (e.g., a few registers or the RAT), and they are copied nearby, very efficiently and in hardware. In all the other schemes, a checkpoint has significant overhead, since larger

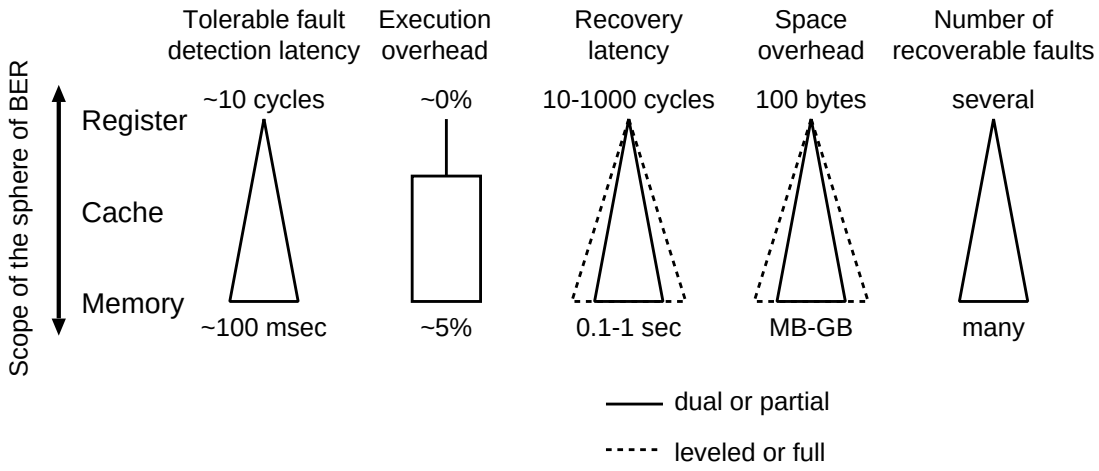

Figure 3. Trade-offs in the different design points of our taxonomy.

chunks of data are copied over longer distances. The overhead of an individual checkpoint tends to increase as we move from cache to memory-level checkpointing. However, designers tune the checkpointing frequency of the different designs to keep the overall execution time overhead tolerable — as an example, the figure shows 5%.

Those designs where individual checkpoints are relatively more expensive take checkpoints less frequently. As a result, both their recovery latency and their space overhead increase (next two triangles). This occurs as we move from cache to memory-level checkpoint. Note that it also occurs as we move from dual to leveled systems. This is because individual checkpoints in leveled systems involve moving data between levels and, therefore, are more expensive. We show this effect with the dotted triangle. A similar effect also occurs as we move from partial- to full-separation designs. In the latter case, there is more data copying.

Finally, as we move from register to cache and to memory-level checkpointing, the number of recoverable faults increases. The reason is two-fold. First, more faults of a given type are supported, thanks to the longer detection latencies tolerated. Secondly, more types of faults are supported (e.g., faults that occur in the new level of the hierarchy). However, additional hardware support is required to support these new faults. Such hardware support is more expensive because it is more spread in the machine.

3 SWICH: Sliding-Window Cache-Level Checkpointing

3.1 Motivation

The previous section showed that the set of trade-offs made by the different schemes is quite different. In this paper, we focus on cache-level checkpointing schemes. These schemes have several positive and negative features.

On the positive side, the hardware of cache-level checkpointing schemes can be integrated relatively inexpensively into modern processor microarchitectures, and can be efficiently used by them. Moreover, technology trends favor more on-chip integration, which enables larger caches. The result is an increase in the fault detection latency of cache-level checkpointing schemes.

Specifically, the main hardware required is new cache line states (Table 1), to mark lines that are related to the checkpoint. The cache replacement algorithm may also need modification to encourage these lines to remain cached. These schemes also need to change the state of groups of cache lines at commit and rollback points. This can be supported by walking the cache tags to change each concerned tag individually, or by adding hardware signals that modify all the concerned tags in one shot. Finally, these schemes also need to 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 [8]. The reason is that the displacement of a dirty cache line triggers a checkpoint. Such a displacement is highly application and cache-organization dependent. This fact makes the performance of a system with a cache-level checkpointing scheme relatively unpredictable.

Another problem of existing implementations of cache-level checkpointing [3, 7] is that the length of the code that they can roll back (the *Rollback Window*) is at times very small. Specifically, suppose that a fault occurs immediately before a checkpoint, and is only detected after the checkpoint has completed — and overwritten the data before the checkpoint. At this point, the amount of code that can be

rolled back is very small, or even zero. The evolution of the rollback window length in this case is illustrated in Figure 4-(a). We can see that, right after each checkpoint, the window is reduced to nil.

3.2 Basic Idea in SWICH

To address the last shortcoming of cache-based checkpointing in an inexpensive manner, we propose SWICH. The novelty of SWICH resides in that it provides 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 of 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 illustrated in Figure 4-(b).

When rollback is required, the two speculative epochs are rolled back. In the worst case of rollback window size (when rollback is required immediately after a checkpoint), the system can roll back at least one speculative epoch. Compared to SWICH, previous schemes have a nil rollback window in the worst case.

In the following, we overview the mechanisms used. In our discussion, we are implicitly referring to actions taken in the L2 cache. In Section 3.5, we include the L1 cache in the design.

3.3 Checkpoint, Commit, and Rollback

We extend each line in the cache with two *Epoch* bits E_1E_0. If the line contains non-speculative (i.e., 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. Note that only dirty lines (those with the Dirty bit set) can have E_1E_0 different 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. Clearing such bits is preferably done 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 are rolled back. This involves restoring the older of the two saved register files and processor state, invalidating all the cache lines whose E_1E_0 bits are not 00, and clearing all the E_1E_0 bits. The operations on the E_1E_0 bits can be done with a one-shot hardware signal. However, since rollback is expected to be infrequent, they can also be done with slower hardware that walks the cache tags and individually sets the bits of the relevant tags. For a write-through L1 cache (Section 3.5), it is simpler to invalidate the whole cache.

3.4 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. Secondly, the cache can only contain a *single* version of a given line. Such a version can be non-speculative 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 the E_1E_0 identifier of its line. 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. Secondly, if the write hits on a non-speculative 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. Thirdly, if the write hits on a speculative line of the same epoch, the update proceeds normally. Finally, if the write hits on a speculative line of the other epoch (say, 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* is started.

Recall that speculative lines cannot be displaced from the cache. Consequently, if space is needed in a cache set, the algorithm first chooses a non-speculative line as 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, one of the just committed lines is chosen as victim for displacement.

There is a fourth condition that leads to an epoch commit. It occurs when the current epoch (say epoch *i*) is about to use all the lines of a given cache set. If we allow it to do it, the sliding window becomes vulnerable: if the epoch later needs an additional line in the same set, we would have

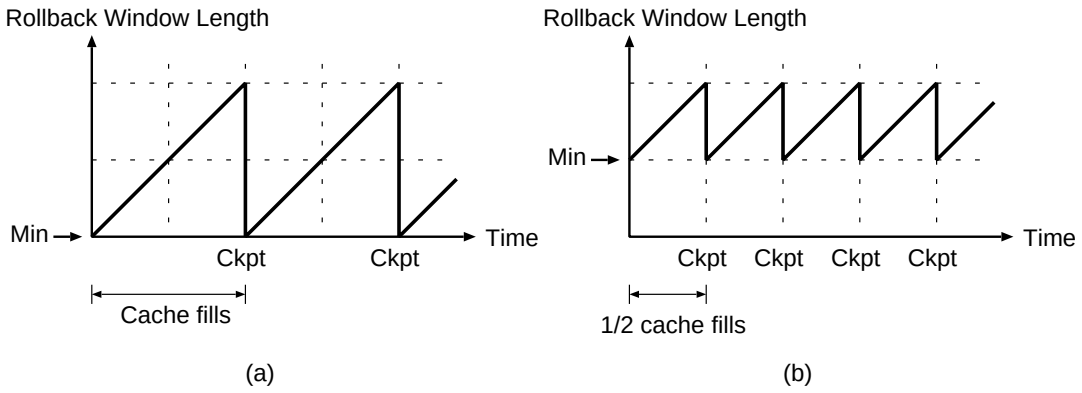

Figure 4. Evolution of the rollback window length with time for Sequoia/CARER (a), and SWICH (b).

to commit *i* (and *i-1*), decreasing the sliding window to size zero. To avoid this, when an epoch is about to use all the lines in a set, we first commit the previous epoch (*i-1*), and then start a new epoch with the access that needs 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 is reduced to zero.

3.5 Two-Level Cache Hierarchies

In a two-level cache hierarchy, the algorithm described is implemented in the L2 cache. It can be shown that the L1 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, the L1 provides the data that it has or that it obtains from L2. On a write hit, the L1 is updated and both the update and epoch bits are propagated to L2. On a commit, no action is taken, while in the rare case of a rollback, the whole L1 is invalidated.

3.6 Multiprocessor Issues

The SWICH improvement is compatible with existing techniques to support cache-level checkpointing in a shared-memory multiprocessor (e.g. [1, 15]). For example, assume that we use Wu *et al.*'s approach [15]. In this case, when a cache needs to 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 is never rolled back to a previous value. The additional advantage of SWICH is that the source cache only needs 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 is not reduced to nil. A similar operation occurs when a cache receives an invalidation for a speculative dirty line. The epoch is committed, the line is provided, and the cache is 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.

4 SWICH Prototype

We implemented a hardware prototype of SWICH using Field-Programmable Gate Arrays (FPGAs). As the base for our implementation, we used Leon2 [6], a 32-bit processor core compliant with the SPARC V8 architecture. Leon2 is in synthesizable VHDL format. It has an in-order, single-issue, five stage pipeline. Most instructions take 5 cycles to complete if no stalls occur. The processor has a windowed register file. 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, the associativity to 8, and vary the cache size. Since 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, namely with (i) cache support for buffering speculative data and roll back, and (ii) register support for checkpointing and rollback. We now describe both extensions in some detail.

4.1 Data Cache Extensions

While the prototype implements most of the design of Section 3, there are a few differences. First, the prototype has a single level of data cache. Secondly, for simplicity, we only allow a speculative epoch to own at most half of the lines in any cache set.

Thirdly, epoch commit and roll back is not performed 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 implement the cache with the synchronous RAM blocks present in the Xilinx Virtex II family of FPGAs. Such memory structures only have the ordinary address and data inputs. They cannot be easily modified to incorporate additional control signals, such as those needed to support one-shot commit and rollback signals.

More specifically, 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). The CWSM is activated when a commit or rollback signal is received. At that point, the pipeline is stalled and the CWSM walks the cache tags. The CWSM has three states (Figure 5-(a)): Idle, Walk and Restore. In Idle, the CWSM is inactive. Walk is its main working state. In this 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 restores the cache controller state 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 non-speculatively. 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.

4.2 Register Extensions

Register checkpointing and restoration is performed using two Shadow Register Files (SRF0 and SRF1). These are two memory structures 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). The RCSM has four states (Figure 5-(b)).

The RCSM is in the Idle state while the pipeline executes normally. When a register checkpoint needs to be taken, it transitions to the Checkpoint state. In this state, the valid registers in the main register file are copied to one of the SRFs. The register files are implemented in SRAM and have two read ports and one write port. This means that we can only copy 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 contents of the register file are restored from the checkpoint. Similarly, this takes about as many cycles as there are valid registers. The Restore state re-initializes the pipeline.

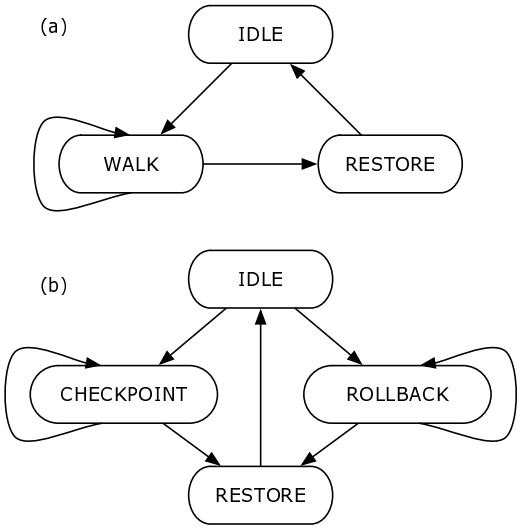

Figure 5. Cache Walk State Machine (a) and Register Checkpointing State Machine (b).

5 Evaluation of the Prototype

5.1 Experimental Infrastructure

5.1.1 Development Board

The processor is part of a system-on-a-chip infrastructure that includes a synthesizable SDRAM controller and 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 [10]. The processor runs at 40 MHz. The board has 64 Mbytes of SDRAM for main memory. Communication with the board and loading of programs in memory are 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 [5]. 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 a set of applications that includes open-source Linux applications and microbenchmarks. We

Application	Description
hennessy	A collection of small, compute-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 files. System-code intensive.
ncompress	Compression and decompression utility. Moderate system code.
ps	Linux Process Status command. Very system code intensive.

Table 2. Applications evaluated.

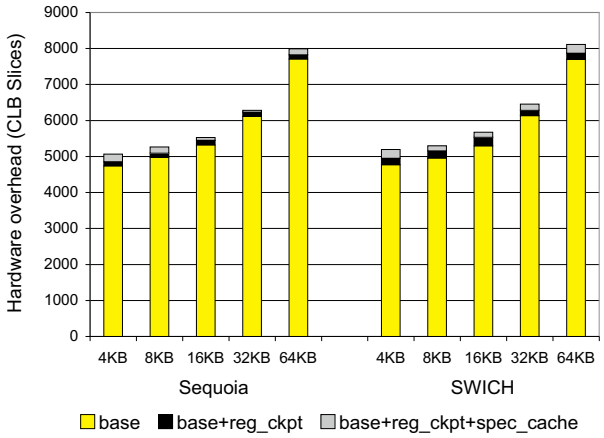

Figure 6. Amount of logic consumed by the different designs and components.

choose codes that exhibit a variety of memory and system code access patterns to give us a sense of how SWICH is affected by workload characteristics. Table 2 shows the applications we use and provides a short description. We run the applications on top of SnapGear Linux. We are in the process of bringing up more standard applications, including SPEC codes.

5.2 Results

We cannot accurately measure the execution overhead or the recovery latency of SWICH. The reason is the implementation limitations explained in Section 4, namely that (i) epoch commit and rollback are implemented with inefficient cache tag walking rather than with an efficient one-shot hardware signal, and (ii) register saving and restoring are implemented in a very inefficient, time consuming manner. These are handicaps that slow down the SWICH implementation and would not be present in a realistic CMOS implementation. Moreover, our FPGA implementation runs

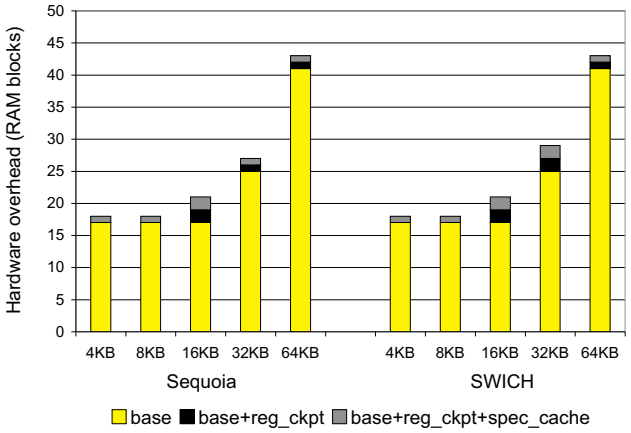

Figure 7. Amount of RAM consumed by the different designs and components.

at only 40 MHz.

Consequently, we focus the evaluation on measuring the hardware overhead of SWICH (Section 5.2.1) and the size of the rollback window (Section 5.2.2), and on roughly estimating the performance overhead and recovery latency of SWICH using a very simple model (Section 5.2.3).

5.2.1 Hardware Overhead

We measure the hardware overhead induced by each of the two components of SWICH, namely the register checkpointing mechanism, and the data cache support for speculative data. We measure the overhead in both logic and memory structures. In our measurements, we only consider the processor core and caches – not the other on-chip modules such as the PCI and memory controllers. As a reference, we also measure the overheads of a cache checkpointing scheme with a single speculative epoch at a time. This system is 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 4 similar slices; each slice includes two 4-input function generators, carry logic, arithmetic logic gates, wide-function multiplexers and two storage elements.

In the figure, the leftmost bars correspond to the Sequoia-like system and the rightmost ones to the SWICH system. Each system has several bars, each one for a design with a different data cache size (4 Kbytes to 64 Kbytes). Each bar is broken down into the logic used by the cache support for speculative data (*spec_cache*), the register checkpointing (*reg_ckpt*), and the original processor (*base*).

Focusing on SWICH, we see that the logic overhead of

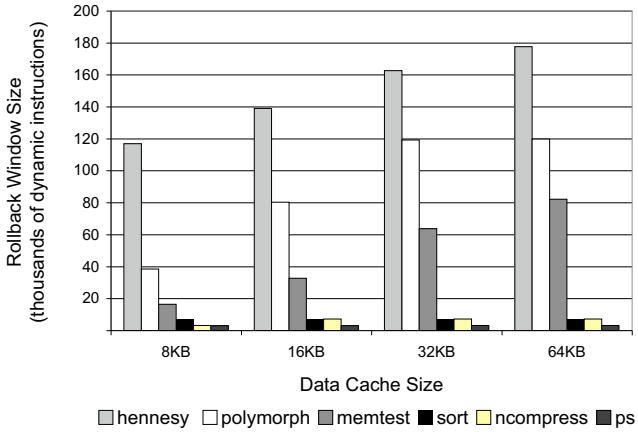

Figure 8. Minimum rollback window size in SWICH (per application averages).

the combined extensions is only 6.5% on average and relatively constant across the range of caches that we evaluate. Consequently, the support that we propose does not use much logic. In addition, we see that the number of SWICH CLBs is only about 2% higher than the Sequoia CLBs. Consequently, 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 are 18 Kbit, dual-port RAMs with two independently-clocked and independently-controlled synchronous ports that access a common storage area. They are used to implement the caches, register files, and other structures.

The figure is organized as Figure 6. It shows that the overhead in memory blocks of the combined SWICH extensions is again small for all the caches that we evaluate. Consequently, the support that we propose consumes few hardware resources. In addition, the overhead in memory blocks of SWICH is again very similar to that of the Sequoia-like system. It largely adds additional register checkpoint storage and additional state bits in the cache tags.

5.2.2 Size of the Rollback Window

We would like to determine the size of the rollback window in SWICH in number of dynamic instructions. Such a number determines how far back can the system roll back at a given time, if a fault is detected. In particular, given Figure 4-(b), we are interested in the *points of minima*, since they represent the cases when only the shortest rollback windows are available. In practice, the points of minima do not all have the same value — Figure 4-(b) shows an ideal case. The actual value of the points of minima depend of the code being executed: the actual memory footprint of the code and the presence of system code. Recall that, in our simple implementation, execution of system code causes the commit of the two speculative epochs (Section 4.1) and, therefore, induces a minimum in the rollback window.

In our experiments, we measure the window size at each of the minima points, and take per-application averages. The results are shown in Figure 8. The figure shows the average window size at minima points for the different applications and cache sizes. The cache sizes examined range from 8 Kbytes to 64 Kbytes. The same information for the Sequoia/CARER systems would show rollback windows of *size zero*.

From the figure, we see that the *hennessy* microbenchmark has the largest minimum rollback window. The application is compute intensive, has little system code, and has a relatively small memory footprint. As the cache increases, the window size increases. For 8-64 Kbyte caches, the minimum rollback window ranges from 120,000 to 180,000 instructions. *polymorph* and *memtest* have more system code or a larger memory footprint, respectively. Their windows are smaller, although they increase substantially with the cache size, to reach 120,000 and 80,000 instructions, respectively, for 64 Kbyte caches. *sort*, *ncompress* and *ps* are system code intensive applications with short rollback windows, irrespective 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 measure the average rollback window size of SWICH and compare it to the Sequoia/CARER-like implementation. Again, we look at four different data cache sizes and six applications. The results are shown in Figure 9. We can see that, except for the system-intensive applications, the average rollback window for SWICH is significantly larger than that of Sequoia/CARER. This is because, as we can see from Figure 8, for SWICH, the rollback window has a non-zero average minimum value. In the case of *ps* and the other two system-intensive applications, there is little difference between the two schemes.

Overall, we conclude that, at least for applications without frequent system code, SWICH can support a large minimum rollback window. This is in contrast to systems like Sequoia and CARER, where the minimum rollback window is zero.

5.2.3 Estimating Performance Overhead and Recovery Latency

A very rough way of estimating the performance overhead of SWICH 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

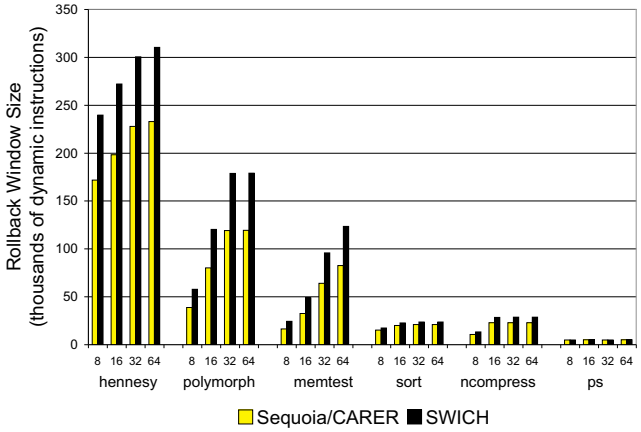

Figure 9. Average rollback window size for Sequoia/CARER and SWICH. The numbers under the bars are cache sizes in Kbytes.

any overheads that the scheme may induce between checkpoints.

To model worst case conditions, we use the average size of the rollback window at minima points (Figure 8) as the number of instructions between checkpoints. Across all applications, this is about 60,000 instructions for 64 Kbyte caches. If we assume an average CPI of 1, this corresponds to 60,000 cycles. On the other hand, a checkpoint can very conservatively take about 256 cycles, as it involves saving registers and asserting a one-shot hardware signal that modifies cache tag bits. Overall, the ratio results in a 0.4% overhead. We stress that this is only a rough estimate.

The recovery latency can be estimated by assuming 256 cycles to both restore the registers and assert a one-shot hardware signal that modifies cache tag bits. In addition, as the processor re-executes, it has to 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.

6 Conclusions

This paper made three contributions. First, it presented a taxonomy and trade-off analysis of different low-overhead checkpointing schemes. Secondly, it described the microarchitecture of a novel cache-level checkpointing scheme called SWICH. The idea behind SWICH is to maintain two speculative epochs at all times, to form a *sliding* rollback window. This enables SWICH to support a large minimum rollback window — compared to the zero-sized minimum rollback window in related previous schemes. Similarly, it enables SWICH to support a larger average rollback window than other schemes.

The third contribution was to implement and evaluate an FPGA-based prototype of SWICH. The evaluation suggested that supporting cache-level checkpointing with sliding window is promising. Specifically, this scheme adds little additional logic and memory hardware to a simple processor. Moreover, for at least the applications without frequent system activity, the scheme sustains a large minimum (and average) rollback window. Finally, execution overhead and recovery latency are expected to be small.

References

[1] R. E. Ahmed, R. C. Frazier, and P. N. Marinos. Cache-aided rollback error recovery (CARER) algorithms for shared-memory multiprocessor systems. In *Proc. 20th Int. Symp. Fault-Tolerant Computing*, pages 82–88, 1990.

[2] H. Ando et al. A 1.3GHz fifth generation SPARC64 microprocessor. *IEEE Journal of Solid-State Circuits*, 38(11):1896–1905, 2003.

[3] P. A. Bernstein. Sequoia: A fault-tolerant tightly coupled multiprocessor for transaction processing. *IEEE Computer*, 21(2):37–45, 1988.

[4] N. S. Bowen and D. K. Pradhan. Processor- and memory-based checkpoint and rollback recovery. *IEEE Computer*, 26(2):22–31, 1993.

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

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

[7] D. B. Hunt and P. N. Marinos. General purpose cache-aided rollback error recovery (CARER) technique. In *the 17th Internationl Symposium on Fault-Tolerant Computing Systems*, pages 170–175, 1987.

[8] B. Janssens and W. K. Fuchs. The performance of cache-based error recovery in multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 5(10):1033–1043, October 1994.

[9] C. Morin, A. Gefflaut, M. Banâtre, and A.-M. Kermarrec. COMA: an opportunity for building fault-tolerant scalable shared memory multiprocessors. In *the 23rd Annual International Symposium on Computer Architecture*, pages 56–65, 1996.

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

[11] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *the 29th Annual International Symposium on Computer Architecture*, pages 111–122, 2002.

[12] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation*. A K Peters, 3rd edition, 1998.

[13] T. J. Slegel et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[14] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *the 29th Annual International Symposium on Computer Architecture*, pages 123–134, 2002.

[15] K. L. Wu, W. K. Fuchs, and J. H. Patel. Error recovery in shared-memory multiprocessors using private cache. *IEEE Trans. Parallel Distrib. Syst.*, 1(10):231–240, April 1990.