Trade-offs in Transient Fault Recovery Schemes for Redundant Multithreaded Processors

Joseph Sharkey[1], Nayef Abu-Ghazeleh[1], Dmitry Ponomarev[1], Kanad Ghose[1], Aneesh Aggarwal[2]

[1] Department of Computer Science
[1] Department of Electrical Engineering
State University of New York at Binghamton
{jsharke, nayef, dima, ghose}@cs.binghamton.edu, aneesh@binghamton.edu

Abstract. CMOS downscaling trends, manifested in the use of smaller transistor feature sizes and lower supply voltages, make microprocessors more and more vulnerable to transient errors with each new technology generation. One architectural approach to detecting and recovering from such errors is to execute two copies of the same program and then compare the results. While comparing only the store instructions is sufficient for error detection, register values also have to be compared to support fault recovery. In this paper, we propose checkpoint-assisted mechanisms for efficient fault recovery that dramatically reduce the number of register values to be compared for detecting soft errors and perform comprehensive investigation of these and other existing recovery schemes from the standpoint of performance, power and design complexity.

1 Introduction and Motivations

The continuous downscaling of CMOS technology leads to smaller transistor feature sizes and the use of lower supply voltages with each new process generation, making the microprocessor chips more vulnerable to soft (or transient) errors. It is projected that the rate at which the transient errors occur will grow exponentially [14] and will soon represent one of the most significant issues in the design of future generation high-performance microprocessors. One popular approach to addressing these challenges is to execute two copies of the same program and then compare the sequence of results generated by each thread [1, 4, 5, 6, 7, 8, 10, 11]. Any discrepancy between the two result sequences indicates the occurrence of a transient error. Such redundant execution can be implemented in the framework of a superscalar processor. However, despite the well-known fact that the execution of just a single thread leaves the processor resources fairly underutilized, running two simultaneous copies while sharing all resources results in very significant performance degradations [1, 6, 9].

Alternatively, a Simultaneous Multithreaded (SMT) processor naturally provides multiple contexts that can be used to execute two copies of the same program (which we call the *main thread* and the *verification thread*) with less impact on performance [4, 7, 8, 11]. Several solutions have been proposed in recent literature to employ SMT support for redundant multithreading, including the schemes that just detect the

transient errors [7] as well as those that support recovery capabilities [11]. The key to avoiding performance loss in the redundant multithreaded environment is to use *staggered execution*, i.e. to delay the execution of the verification thread by a number of instructions (defined as *slack* in the rest of the paper) behind the main thread. With growing memory latencies, a larger amount of slack between the two threads can help in hiding the memory access delays experienced by the main thread. To take advantage of the staggered execution, the slack is built and maintained during the normal execution and it is consumed (the verification thread catches up with the main thread) on L2 cache misses. Another advantage of maintaining a sufficient amount of slack is that the actual branch outcomes supplied by the main thread can be used by the verification thread instead of branch predictions. This, in turn, eliminates the execution of the wrong-path instructions from the verification thread, further increasing the execution efficiency and reducing the contention for the use of shared datapath resources.

The basic scheme to provide the transient fault *detection* capabilities in an SMT processor, called SRT (Simultaneously and Redundantly Threaded processor) was introduced in [7]. In SRT, only the results (addresses and data) of the store instructions are compared, because any faults in the registers eventually propagate through the dependency chains to a store. However, if the capability to recover from such faults is also essential, then not only the values to be stored into the memory, but also all values written into the register file need to be verified. Otherwise, the recovery to a precise verified state following a transient error may be impossible, as such a state may never exist. In this paper, we perform a comprehensive study of the trade-offs in the design of fault recovery schemes, encompassing the issues of performance, energy and design complexity. These schemes include the previously published methods, as well as the ones that are proposed here. We begin by describing the architecture of the baseline SRT machine used for fault detection.

2. Baseline Architecture

The baseline redundant multithreaded processor that we use for our evaluations is based on the SRT model of [7] for transient fault detection. We assume that both main and verification threads perform separate register allocations, so that the register file is also protected. To introduce the slack between the execution of the two threads, we implemented the slack fetch mechanism described in [7]. The address and data of each store instruction are verified before the store is permitted to update the memory. To verify the address and data of store instructions, an ordered non-coalescing queue, called the *store buffer* (SB) is used, as in [7]. The SB is shared between the threads to synchronize and verify store values as they retire in program order. Data from the store buffer is forwarded to subsequent loads only when the store is retired in the thread issuing the load. The work of [7] proposes two alternatives for the input replication of load data. We implement the *load value queue* (LVQ) – which was shown to provide superior performance [7]. When a load commits from the main thread, it writes both its address and data into the LVQ. Subsequently, when the same load issues in the verification thread, the address is verified and the data is read from the LVQ (i.e., the verification thread does not access the D-cache). This increases performance because the verification thread does not experience cache misses and

does not compete for the cache ports. Finally, to eliminate the wrong-path instructions in the verification thread, we use the *branch outcome queue* (BOQ) [7]. This buffer delivers the committed branch outcomes from the main thread to the verification thread, effectively providing near oracle branch prediction for the verification thread (except in the case where a transient fault causes an incorrect branch resolution in the main thread).

3. Transient Fault Recovery Schemes for SMT

In this section, we describe several possible transient fault recovery schemes that provide recovery capabilities on top of SRT.

3.1. SRT+: Augmenting SRT with Full Register Checking

The first technique that we consider is a trivial augmentation to SRT to check all register values in addition to the data and the addresses of all store instructions. To reduce the pressure on the register file, this requires the addition of a queue (called Register Value Queue – RVQ), where the register results produced by the main thread are written after they are committed. These results are removed from the RVQ only after they are verified by the trailing thread. In this scheme, all register values are checked and an instruction that caused the transient fault can be identified precisely at the earliest possible opportunity. However, a large RVQ is needed to support sizable slack and significant energy is expended in the course of verifying all of the produced register values – those that have to be written to the RVQ, read from it, and compared.

3.2. SRTR and Dependence-Based Checking Elision (DBCE)

The next technique that we examine is called SRTR (SRT with Recovery) and it was introduced in [11]. In addition to checking the store instructions, the SRTR scheme also validates register values, but in contrast to SRT+ it does so selectively. To reduce the pressure on the RVQ and the number of verifications, the authors of [11] also proposed Dependence-Based Checking Elision (DBCE) – a mechanism to limit verifications to only the instructions at the end of short dependency chains, avoiding (or eliding) the verification of the other register values. As reported in [11], about 35% of all register checks are avoided (elided) on the average across SPEC 95 benchmarks using the DBCE scheme.

The original SRTR scheme requires that the result verification occurs prior to instruction commitment (using the writeback-to-commit time), thus putting a limit on the amount of slack that can be maintained. To accommodate a relatively short slack, the SRTR scheme uses the branch predictions (rather than the branch outcomes as in SRT) from the main thread to feed to the verification thread. As a result of the small slack and the use of branch prediction in the verification thread, the SRTR scheme has some performance overhead compared to the SRT design and also incurs some additional changes in the datapath mainly stemming from the need to support speculative instructions in the verification thread. The performance challenges faced

by the SRTR scheme will only be exacerbated in the environments with lower branch prediction accuracies and/or D-cache hit rates as well as higher memory latencies.

We observe that it is possible to move the verification actions in the SRTR/DBCE scheme to the post-commit stages by committing the instructions from the main thread and establishing the RVQ entries at that time, just as in SRT+ scheme. The key is not to allow the commitment of any instruction from a dependency chain in the verification thread until the entire chain is verified. The state of the verification thread then can be used to restart the execution following the detection of a fault. This modification allows the DBCE scheme to be used with larger slack and use branch outcomes instead of branch predictions to avoid the execution of wrong-path instructions by the verification thread.

3.3. Checkpoint-Assisted Fault Recovery Schemes

In this paper, we propose novel schemes to further reduce the number of register values that need to be verified to guarantee recovery to a safe state compared to what is proposed by the DBCE scheme. The philosophy of the DBCE is to support a rollback to the latest checked and committed instruction following the detection of a fault and to begin the re-execution from that point. While such an approach completely avoids unnecessary re-executions of already verified instructions, the datapath complexities and performance overheads involved are non-negligible. In essence, from the standpoint of precise state reconstruction, the SRTR scheme treats transient faults like branch mispredictions or exceptions because it maintains the results of all unchecked instructions, just as the results of all speculative instructions are maintained for branch misprediction recovery or interrupt handling.

However, even in current and future technologies, the absolute rate at which transient faults will occur is very low, several orders of magnitude smaller than, for example, the rate of branch mispredictions or exceptions. Therefore, it is unnecessary to start the re-execution at the exact instruction that caused transient fault; even if the rollback occurs to a point which requires several tens of thousands of instructions to be re-executed, there is almost no impact on performance. The key question here is not how far to rollback and how many instructions to re-execute (within reasonable distance), but how to guarantee that a precise and completely verified register and memory state is always available and can be constructed at any point. In the rest of this section, we describe two checkpoint-based mechanisms to facilitate such a recovery. After the detection of an error, the processor state is rolled back to a complete and fully-verified register and memory state checkpoint and the execution restarts.

3.3.1 Lifetime-Based Checking Elision (LBCE)

It has been noticed by several researchers that most of the register instances in a datapath are short-lived [19]. A value produced by the instruction X is *short-lived (SL)* if the architectural register allocated as a destination of X has been renamed again before the value generated by X is committed. In [20], it was shown that about 84% of all produced values are short lived. Using this notion of short-lived values,

[20] proposes *lifetime-based checking elision* (LBCE) in which the verifications of *control-independent short-lived* (CISL) values are avoided. Only the non-CISL results are saved within the RVQ after the instruction commitment and are verified against similar values produced by the verification thread.

To support the capability to recover to a precise and completely verified state following a detection of a transient fault, LBCE relies on the creation of the periodic register and memory state checkpoints. To buffer a large number of store instructions between two consecutive checkpoints, we use the approach described in [21] and also used in a few others works. The memory updates received between two consecutive checkpoints are stored within the local cache hierarchy, but their propagation to the main memory is avoided until it is safe to do so. Each cache line updated in this manner is marked as volatile, using one extra bit for each cache line. When a processor needs to rollback to a checkpoint, all cache lines marked volatile are invalidated using a gang-invalidate signal. When the new checkpoint is created, all volatile bits set since the creation of previous checkpoint are cleared. A recent paper [23] also describes how to correctly incorporate caches with the volatile lines into a multiprocessor system.

Since transient faults are very infrequent events, we can create checkpoints at very large intervals. In fact, a checkpoint can be created on demand, when one of the sets within the cache has all its lines in Volatile status and a cache miss occurs that targets this set. At this point, the creation of a new checkpoint is initiated and, once the checkpoint is created, the volatile bits can be cleared. However, as the percentage of volatile lines in the cache increases, the victim selection algorithm becomes less flexible (the volatile lines cannot be replaced). In the worst case, this effectively transforms the cache into direct-mapped structure and degrades the cache hit rates. In order to avoid such performance degradations caused by the lower D-cache hit rates, we also force the checkpoint creation every 100000 instructions. Therefore, 100000 instructions are re-executed after transient fault detection in this scheme in the worst case. In the result section, we quantify the percentage of checkpoints created for these various reasons. We also show that the average number of instructions between two consecutive checkpoints is generally very large. A recent paper [24] also showed that in commercial workloads the I/O operations could occur more frequently, effectively requiring the creation of a checkpoint at that instant. To support these situations, in the results section we also evaluate the performance of the LBCE scheme with smaller checkpointing periods, as low as 500 instructions.

For more details of the LBCE technique, including the hardware implementation to detect the CISL values, we refer the reader to [20].

3.3.2 An RVQ-Free Recovery Scheme (RVQ_F)

We will now describe the checkpoint-assisted recovery scheme that completely eliminates the RVQ from the datapath. In this scheme, the decision to create a checkpoint can be triggered at the time of committing an arbitrary instruction from the main thread. At this point, the main thread is stalled and the verification thread is allowed to completely catch up (consume the slack). At that time, the contents of the architectural register state from both threads can be compared against each other, and if any mismatch occurs, then a transient fault is detected. Otherwise, new checkpoints

of the register file and commit-time rename table can be created. Also, the volatile bit in the cache can be cleared.

Table 1: Comparison of the key features of the transient fault recovery schemes. Quantitative comparisons are provided in the results section.

	SRT+	SRTR	LBCE	RVQ_F
Checkpoints required	No	No	Yes	Yes
RVQ required	Large	Medium	Small	None
Additional Logic Needed	None	Track and form dependency chains	Detect short-lived values	None
Transient-Fault Detection Latency	Short	Short	Short to medium	Large
# of register verifications	All register values	~65% of register values	~ 30% of register values	Only on checkpoint creation
Useful work lost on every fault	None	None	Small to medium	High
Reasons for stalling the main thread	RVQ is full	RVQ is full	RVQ is full	Checkpoint creation

While this scheme simplifies the datapath compared to the LBCE technique from the previous section, it incurs some performance overhead. First, the main thread needs to stall during the checkpoint creation – that is not required by the LBCE. Second, the bulk-comparison of the architectural registers will require a number of cycles to be wasted: for example, for 64 architectural registers (as in the Alpha ISA), the comparisons will consume 16 cycles (if 4 comparisons can be performed per cycle). For small checkpointing periods, these overheads can be significant; we evaluate the sensitivity of these schemes to the checkpointing frequency in the results section. Finally, the RVQ_F scheme is likely to delay the detection of transient errors, as the detection can only occur during the checkpoint creation. In the next section, we compare all of the described techniques in terms of their performance, energy consumption, complexity and other metrics. Table 1 summarizes the key features of the four transient fault recovery schemes examined in this paper. A detailed quantitative comparison of the schemes follows later.

Table 2: Simulated processor configuration.

Parameter	**Configuration**
Machine width	4-wide fetch, 4-wide issue, 4-wide commit
Window size	64 entry issue queue, 64 entry load/store queue, 128-entry ROB
Pipeline Depth	5 cycles fetch to dispatch, 3 cycles issue to execute
Function Units and Lat (total/issue)	4 Int Add (1/1), 2 Int Mult (3/1) / Div (20/19), 2 Load/Store (2/1), 4 FP Add (2), 2 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
Phys. Registers	300 combined integer and floating-point
L1 I–cache	64 KB, 4–way set–associative, 32 byte line
L1 D–cache	64 KB, 4–way set–associative, 32 byte line
L2 Cache unified	1 MB, 8–way set–associative, 128 byte line
Memory latency	100 cycles
TLB	64 entry (I), 128 entry (D), fully associative

4 Simulation Methodology

For estimating the performance impact of the schemes described in this paper, we used M-Sim [12] – a significantly modified version of the Simplescalar 3.0d simulator

[1] that separately models pipeline structures such as the issue queue, re-order buffer, and physical register file, both for superscalar and SMT machines [5,6]. The SRT model described in Section 2 was implemented in this framework. The details of the studied processor configuration are shown in Table 2.
We simulated a total of 24 integer and floating point benchmarks from the SPEC 2000 suite [3], using the precompiled Alpha binaries available from the Simplescalar website [1]. Predictors and caches were warmed up for the first 1 billion instructions and the statistics were gathered for the next 100 million instructions.

5 Results and Discussions

Figure 1 compares the performance of the transient fault recovery schemes that rely on the RVQ. Results are presented in terms of harmonic means across all simulated SPEC 2K benchmarks. The first variation is the SRT scheme which only supports fault detection – this represents an upper bound on the performance, as there is no recovery overhead. The other lines correspond to the SRT+, the DBCE scheme, and the LBCE scheme with various checkpointing periods. The number next to the LBCE label in the ledged signifies the checkpointing period (number of instructions) used for the corresponding configuration.

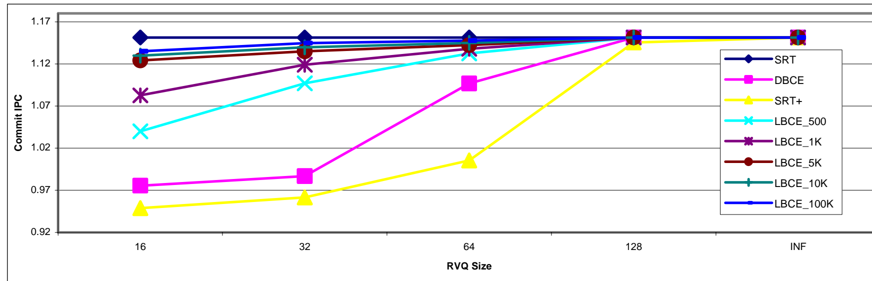

Figure 1: Harmonic mean of commit IPC for various redundant multithreaded architectures for various sizes of the RVQ.

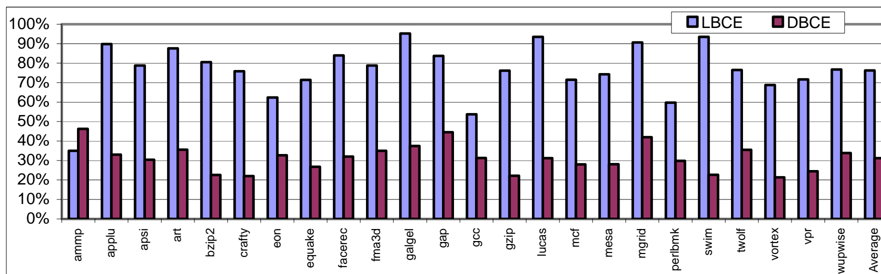

Figure 2: Percentage of register verifications elided using the LBCE and DBCE schemes.

For these experiments, the target slack of 256 instructions (shown to be optimal in [7] and also confirmed by our experiments) was used. Because not all instructions are verified through the RVQ (loads, stores and branches are not), the performance saturates for all schemes at the RVQ size of 128 entries, with the saturation in the

LBCE scheme occurring at much smaller RVQ sizes. The SRT+ scheme results in significant performance losses compared to simple SRT at smaller RVQ sizes. For example, the average performance losses are 21%, 19% and 15% for the RVQ sizes of 16, 32 and 64 entries respectively. The DBCE reduces the performance overhead of SRT+ and lowers the performance degradations to 18%, 16%, and 4.9% respectively for 16, 32 and 64-entry RVQ compared to the SRT+ design. Next, the LBCE scheme with the small checkpointing period of 500 lowers these percentages further to 10%, 5% and 1.6%. Finally, the LBCE scheme with a large checkpointing period of 100,000 instructions lowers these percentages to 1.4% 0.5% and 0.3%. Notice that the LBCE scheme with larger checkpointing periods provides better performance as the overhead of checkpoint creation is small. In summary, a 16-entry RVQ with the LBCE scheme provides almost the same performance as the SRT without any recovery overhead or as the SRT+ with 128-entry RVQ.

The reason for the performance improvements in both the DBCE and the LBCE schemes for small RVQ sizes is that many of the register verifications are elided and therefore fewer instructions require entries in the RVQ. Figure 2 presents the percentage of register verifications that are elided using the LBCE and DBCE schemes. While the LBCE scheme elides 76.1% of the verifications, the DBCE scheme elides about 32% of the verifications for the Spec2000 benchmarks (the results in [11] showed 35% for the Spec95 benchmarks). The larger percentage of register value checks that are elided by LBCE are manifested in higher IPCs.

Table 3: Number of cycles when the leading thread stalls because the RVQ is full.

	16-entry RVQ	32-entry RVQ	64-entry RVQ	128-entry RVQ	256-entry RVQ
SRT+	81700029	79098756	72103164	32808131	281531
DBCE	75477795	72893922	53269057	8252604	38
LBCE_10K	36841417	24369222	6297368	315680	0

Table 4: Average number of read and write ports to the RVQ used by the various schemes.

	# RVQ write ports used per cycle	# RVQ read ports per cycle
SRT+	2.9447	2.9447
DBCE	2.0505	2.0505
LBCE_100K	0.4898	0.4898
LBCE_10K	0.5045	0.5045
LBCE_5K	0.5219	0.5219
LBCE_1K	0.6417	0.6417

The size of the RVQ has a profound influence on the overall performance of the schemes that require a RVQ, as shown in Table 3. Whenever the RVQ is full, the main thread is stalled and the verification thread is run, preventing further progress of the main thread momentarily. As seen from Table 3, the LBCE scheme has a significant advantage over the others that use a RVQ, as it stores only the non-CISL values. At about an RVQ size of 256 entries, both DBCE and LBCE avoid any stalls of the main thread. In contrast, some stalls still occur for the SRT+ scheme at this RVQ size. Therefore, a smaller RVQ size is sufficient for the LBCE scheme to provide similar performance.

Next, we examine the impact on dynamic power dissipation within the RVQ of our technique. We compare two configurations that achieve the same performance, specifically a 32-entry RVQ with LBCE scheme and 128-entry RVQ with SRT+ scheme. The savings in dynamic power of LBCE scheme comes from two sources.

First, much fewer access to the RVQ are performed because 76% of the checks are elided, and second the size of the RVQ is significantly smaller. Combined, these two artifacts result in 89.1% savings in dynamic power within the RVQ compared with the SRT+ design. Of course, additional power would be dissipated in the auxiliary datapath structures required by the LBCE scheme, which will somewhat lower these reported savings. However, if the point of comparison is the DBCE mechanism, then it also requires additional power to detect and form the dependency chains in both threads. A more detailed power related analysis of these mechanisms is beyond the scope of this paper.

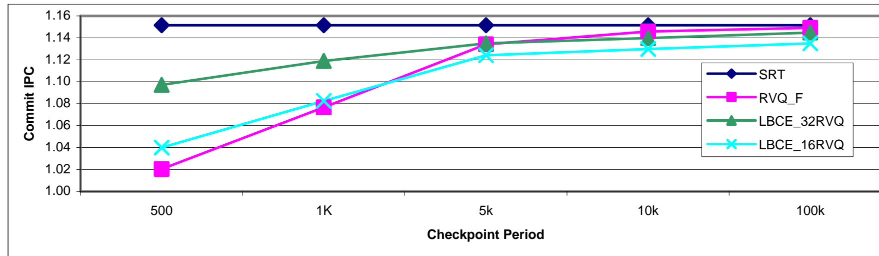

Figure 3: Harmonic mean of commit IPC for various transient fault recovery schemes for various checkpointing intervals.

Additionally, because many of the register verifications are elided, fewer reads and writes to the RVQ are performed each cycle with the LBCE scheme compared to the DBCE and SRT+ techniques. Table 4 presents the average number of read ports and write ports used per cycle to the RVQ for the various transient fault recovery schemes. The SRT+ technique, with allocates an RVQ entry for each and every register value, uses nearly 3 read ports and 3 write ports each cycle on average. Comparatively, the DBCE scheme uses only 2 read ports and 2 write ports on average each cycle and the LBCE technique uses less than one. This allows for a reduction in the number of ports to the RVQ with the LCBE scheme in addition to the reduction in RVQ size – which provides additional energy and power savings.

Now, we examine the checkpoint-based transient fault recovery solutions. Figure 3 presents the harmonic mean of commit IPC for the RVQ_F and LBCE schemes (the only two schemes that rely on checkpointing) for various checkpointing periods. The SRT scheme that does not provide recovery is also shown for comparison. For the small checkpointing periods, LBCE outperforms RVQ_F because the overhead of the frequent checkpoint creations offsets the advantages offered by the RVQ_F scheme. For example, the LBCE scheme with a 32-entry RVQ outperforms the RVQ_F scheme by 4% for a checkpointing period of 1000 instructions and 8% for the checkpointing period of 500 instructions. On the other hand, for large checkpointing periods, the RVQ_F scheme provides better performance. For the period of 100K instructions, the RVQ_F scheme outperforms the LBCE scheme by 1.5%.

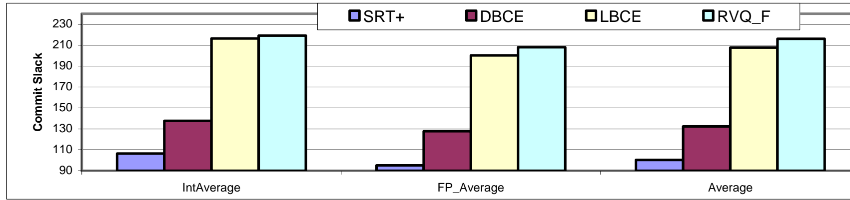

Figure 4: Effective slack length measured in number of instructions at commitment.

The RVQ_F is quite efficient for large checkpointing frequencies because it elides most of the register checks (other than the ones that are needed for checkpoint creation) by the nature of the scheme. For example, for checkpointing period of 100000 instructions, 99.8% of all register verifications are elided. For 500-instruction checkpointing period, the percentage of elided checks is about 80%.

The next metric that we examine is the effective slack length as measured at commit time. The results for the 64-entry RVQs are presented in Figure 4. For this configuration, the LBCE scheme achieves a slack of 207 instructions, on the average – more than twice that of the processor with the basic SRT+ which achieves a slack of only 101 instructions. The DBCE scheme achieves the slack of 135 instructions. These results show that the LBCE technique can maintain a large slack, and take advantage of it, with a small RVQ size. In fact, the amount of the effective slack in the LBCE scheme even with 16-entry RVQ is almost the same as the effective slack of the SRT+ scheme with infinite RVQ (again, the results of Figure 1 can be used to understand why that is the case). Finally, the RVQ_F scheme achieves an average slack of 210 instructions.

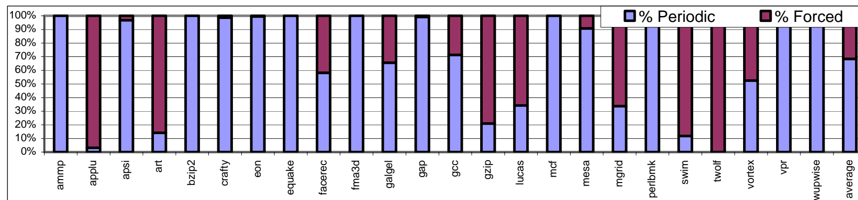

Figure 5: Breakdown of the percentage of checkpoints created periodically versus the percentage of forced checkpoints due to cache behavior for the LBCE and RVQ_F schemes. Results are presented for the checkpointing period of 100000 instructions.

Next, we evaluate the impact of the checkpointing mechanism used by LBCE and RVQ_F in order to support recovery from transient faults. Recall that there are two triggers for the creation of checkpoints in these schemes. Checkpoints are created periodically, or when required due to the absence of non-volatile data in the cache set for victim selection. Figure 5 presents the data on the percentage of checkpoints created due to each of these triggers. As seen from the graph, 69% of the created checkpoints are induced periodically. The percentage of checkpoints that are created due to the absence of non-volatile lines in the accessed set of the cache is relatively small on the average, but can be quite high for the memory bound programs. For example, applu, art, swim, and twolf all experience high levels of memory traffic and therefore incur more such checkpoints.

It is conceivable that the use of volatile bits in the cache can somewhat degrade the cache hit rates because of the additional constraints imposed on the cache replacement policies. However, our results indicate that this impact is minimal. On the average, the L1 D-cache hit rates decreased from 94.6% to 94.5%, and the largest decrease was 2.3% observed on ammp.

6 Related Work

A popular approach for concurrent error detection and recovery is to execute two copies of the same program and then compare the results [1,4,5,6,7,8,10,11]. Ray, Hoe, and Falsafi [6] propose mechanisms for performing such redundant execution within a superscalar processor. Smolens et. al. [9] study the performance impact of redundant execution and identify the various bottlenecks that limit the performance in such environments. The DIVA design of [1] supplemented the out-of-order core with simple in-order checker logic. The fault-tolerant architectures in [4,7,8,11] use the inherent hardware redundancy in SMT and CMP architectures for concurrent error detection. While the SRT scheme described in [7] only aims at detecting transient faults using the SMT support, the follow up study of [11] augments the work of [7] by adding the recovery capability. The resulting scheme, called SRTR (SRT with Recovery) is perhaps the closest in spirit to the proposal. We extensively discussed the SRTR scheme and contrasted it to techniques proposed here throughout the paper. RMT explored the design space of using multithreading for fault detection [15], and was extended by CRTR [16] to provide fault recovery using CMPs. The concept of partial soft error coverage was introduced in [5], where the redundant execution is only performed during the low-ILP phases of the main program, when the resources are sufficiently underutilized. In [2], the execution of the redundant thread only happens when the main thread experiences the L2 cache miss or the verification buffer is full. Several industrial designs support fault tolerance. The Compaq NonStop Himalaya [12] employs off-the-shelf microprocessors in lock-step fashion and compares the outputs every cycle. The IBM S/390 [18] uses replicated, lock-stepped pipelines within the processor itself.

7 Summary and Concluding Remarks

The choice of the best transient fault recovery scheme is dictated by the checkpointing interval as well as datapath complexities that can be tolerated. We can expect aggressive modern out-of-order processors to use checkpoint-based recovery mechanisms. Some of the schemes studied in this paper assume the existence of such a facility. There is always a tradeoff between the performance, the complexity, and the energy consumption that guide the choice of the soft error detection and recovery scheme. The main conclusions of our study, in the light of such considerations, are as follows.

If large checkpointing intervals can be tolerated, then the RVQ_F scheme provides the best performance because of the least number of register values comparisons –

only the architectural register values need to be compared at the time of checkpoint creation. Furthermore, RVQ_F scheme eliminates the need for an RVQ and all associated overhead. However, at smaller checkpointing intervals, the LBCE mechanism is more attractive because it achieves better performance for a smaller RVQ size relative to SRT+ and DBCE. Furthermore, the data on the usage of read and write ports shows that the LBCE technique can not just use a smaller RVQ compared to SRT+, but it can also use fewer register file ports, thereby reducing the overall power dissipation (and the overall complexity) of the verification logic. If a large RVQ can be supported, then schemes that do not rely on checkpointing, such as SRT+ and DBCE, are both reasonable choices.

References

1. T. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," Proc. Micro-32, 1999.
2. Qureshi, M., et al, "Microarchitecture-Based Introspection: A Technique for Transient Fault Tolerance in Microprocessors", in DSN 2005.
3. J. G. Holm, and P. Banerjee, "Low cost concurrent error detection in a VLIW architecture using replicated instructions" Proc. ICPP-21, 1992.
4. M. Gomaa, et. al., "Transient-Fault Recovery for Chip Multiprocessors," Proc. ISCA-30, 2003.
5. M. Gomaa, T.N.Vijaykumar, "Opportunistic Transient Fault Detection", ISCA 2005
6. J. Ray, J. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," Proc. Micro-34, 2001.
7. S. Reinhardt, and S. Mukherjee, "Transient fault detection via simultaneous multithreading," Proc. ISCA-27, June 2000.
8. E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," Proc. 29th Intl. Symp. On Fault-Tolerant Computing Systems, 1999.
9. J.Smolens, et. al., "Efficient Resource sharing in Concurrent error detecting Superscalar microarchitectures ," Proc. Micro- 37, 2004.
10. K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," In Proc. Micro-33, December 2000.
11. T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," Proc. ISCA-29, 2002.
12. J. Sharkey. "M-Sim: A Flexible, Multi-threaded Simulation Environment." Tech. Report CS-TR-05-DP1, Department of Computer Science, SUNY Binghamton, 2005.
13. D. Tullsen, et al. "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor." in Proc International Symposium on Computer Architecture, 1996.
14. P. Shivakumar, et al. "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic", in Proc DSN, 2002.
15. S. Mukherjee, et al. "Detailed Design and Evaluation of Redundant Multithreading Alternatives", in Proc ISCA 2002.
16. M. Gomaa, et al. "Transient-fault Recovery for Chip Multiprocessors" in Proc ISCA 2003.
17. Compaq zComputer Corporation, "Data Integrity for Compaq Non-Stop Himalaya Servers", 1999.
18. T. Slegel, et al. "IBM's S/390 G5 Microprocessor Design", IEEE Micro, 1999.
19. D. Ponomarev, et al., "Reducing Datapath Energy through the Isolation of Short-Lived Operands", Proc. PACT 2003.
20. N. Abu-Ghazeleh, et al., "Exploiting Short-Lived Values for Low-Overhead Transient Fault Recovery", Proc. ASGI 2006.
21. J. Martinez, et al., "Cherry: Checkpointed Early Resource Recycling in Out-of-Order Processors", Proc. MICRO 2002.
22. O.Ergin, et al., "Increasing Processor Performance through Early Register Release", Proc. ICCD 2004.
23. M. Kirman, et al., "Cherry-MP: Correctly Integrating Checkpointed Early Resource Recycling in Chip Multiprocessors", Proc. MICRO 2005.
24. J. Smolens, et al, "Fingerprinting: Bounding Soft-Error Detection Latency and Bandwidth", Proc. ASPLOS 2004.