Exploiting Short-Lived Values for Low-Overhead Transient Fault Recovery

Nayef Abu-Ghazaleh, Joseph Sharkey, Dmitry Ponomarev, Kanad Ghose
Department of Computer Science
State University of New York at Binghamton, Binghamton NY
{nayef, jsharke, dima, ghose}cs.binghamton.edu
http://caps.cs.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 such errors and recovering from them is to execute two copies of the same program and then compare the results. In this paper, we propose a technique to dramatically reduce the performance and energy overhead of result verification by exploiting the observation that a large percentage of generated result values in a datapath are short-lived. Our scheme avoids verification of short-lived values without impeding transient fault detection and recovery capabilities.

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. These transient errors, also known as "single event upsets", occur for various reasons, for example when cosmic alpha particles energize or discharge internal nodes of logic or SRAM bitcells, resulting in incorrect operation. It is projected that the rate at which the transient errors occur will grow exponentially [26] 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 compare the results [1, 7, 13, 14, 15, 16, 21, 22]. A mismatch in the results produced by the two instruction streams indicates a transient fault. Such redundant execution can be performed 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, 14, 20].

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 [7, 15, 16, 22]. Several solutions have been proposed in recent literature to employ SMT support for redundant multithreading, including the schemes that just detect the transient errors [15] as well as those that support recovery capabilities [22].

The key to avoiding performance loss in the redundant multithreaded environment is to use *staggered execution*, i.e. to run the verification thread 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. Indeed, if a sufficient number of instructions from the verification thread are available for processing while the L2 cache miss request from the main thread is being serviced, then the number of cycles lost due to the memory access can be minimized, if not avoided altogether. Otherwise, if the two threads are executing roughly the same instructions at the same time, then they both will stall on a long-latency cache miss. 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) was introduced in [15]. 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, it may be impossible to recover to a precise and fully verified state following transient fault detection.

To provide recovery capabilities on top of SRT, a technique called SRTR (SRT with Recovery) was introduced in [22]. This is the starting point for our designs. In addition to checking the store instructions, the SRTR scheme also validates register values. To reduce the pressure on the register file, a separate queue (called RVQ – Register Value Queue) was used in [22] to implement register verification actions. To reduce the pressure on the RVQ and the number of verifications, the authors of [22] also proposed Dependence-Based Checking Elision (DBCE) – a mechanism to limit verifications to only the instructions at the end of short dependency chains. As reported in [22], about 35% of all register checks are elided on the average across SPEC 95 benchmarks using the DBCE scheme.

The SRTR technique does not allow an instruction to commit before its result is verified thus imposing a limit on the amount of slack. 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, the SRTR scheme has some performance overhead compared to the SRT design and also incurs datapath changes stemming mainly 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 DBCE scheme to the post-commit stages to support larger slack and avoid the execution of wrong-path instructions in the verification thread. This can be accomplished by committing the instructions from the main thread *before verification* and establishing the RVQ entries at that time. 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. For comparing the technique proposed in this paper with SRTR, we use this mechanism.

In this paper, we propose a transient fault recovery scheme that further reduces the percentage of register values that need to be verified, even compared to the DBCE scheme, without losing the capability to detect the faults and recover from them. Our technique relaxes the requirement that, following the detection of a fault, the processor must roll back to the latest instruction that completed execution without a fault. While such an approach completely avoids unnecessary re-executions of already verified instructions, the datapath complexities and performance overhead 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 and likely to be several orders of magnitude smaller than the branch misprediction rate or the rate of 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 consideration here is not the extent of the roll back and the number of instructions to re-executed (within reasonable distance), but rather the facilities needed to guarantee that a *precise and completely verified* register and memory state is always available at any point.

We leverage the support for register and memory state checkpointing and propose a technique to dramatically reduce the number of register values that have to be verified (even when compared with the DBCE scheme). Our technique (called LBCE – *Lifetime Based Checking Elision*) exploits the fact that a majority of values (84%) generated in the datapath are *Short-Lived* (SL), i.e. the architectural registers targeted by these values are renamed again before the values are committed. Furthermore, in 76% of the cases the fact that a value is short-lived does not depend on the control flow (we call these values CISL – "control-independent short-lived values"). That is, either there are no branches between the instruction producing a SL value and the instruction renaming the same destination register, or all such branches are correctly resolved by the time the instruction commits. We propose to avoid the verification of CISL instructions and the allocation of RVQ entries to them, without impeding the ability to rollback to a precise state following transient fault detection.

Our technique dramatically reduces the verification overhead in the course of the predominant fault-free execution periods (76% of register checks are avoided without losing the recovery capability), at the cost of an increase in the re-execution overhead (we allow at most 100000 instructions worth of work to be re-executed following transient fault detection).

We demonstrate that if the RVQ is a performance bottleneck, then significant performance increases are possible compared to both the SRT scheme augmented with full register verification and the

DBCE scheme with a similarly sized RVQ. Otherwise, a much smaller RVQ can be used for the same performance, resulting in a substantial reduction in both the datapath power and complexity. Furthermore, we also report significant reduction in the power dissipations incurred during the verification process.

The rest of the paper is organized as follows. Section 2 describes the baseline organization of a redundant multithreaded machine. We present our technique in Section 3. The simulation methodology is described in Section 4 and the results are presented and discussed in Section 5. We review the related work in Section 6. Finally, we offer our concluding remarks in Section 7.

2. Baseline Architecture

The baseline redundant multithreaded processor that we use for our evaluations is based on the SRT model of [15] for detection and the SRTR [22] model for recovery. 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 [15] for arbitrating between the main and verification threads at the fetch stage. This mechanism relies on a signed counter that is incremented when instructions commit from the main thread and decremented when instructions commit from the verification thread. This counter is initialized and reset to the target value of slack, and is added to the instruction count of the verification thread before the ICOUNT [25] mechanism is invoked. Such a mechanism essentially favors the main thread until the desired amount of slack is accumulated. On L2 cache misses or other events that block the progress of the main thread, the accumulated slack is consumed by fetching and executing instructions from the verification thread.

Our model implements the *Register Value Queue* (RVQ) to buffer register values awaiting verification after their production or commitment from the main thread. The actual verification can occur either before or after commitment, for the purposes of this study we assume post-commit verification in SRTR, as described earlier in Section 1.

The address and data of each store instruction are also 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 [15]. 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 [15] proposes two alternatives for the input replication of load data. We implement the *load value queue* (LVQ) – which was shown to provide superior performance [15]. 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 the 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) [15]. 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). With this technique, a deviation in the control flow paths between the main and verification threads is signaled by a branch misprediction in the verification thread.

The entries in all of these queues are established by instructions in the main thread at the time of commitment. If there are no free entries available to such an instruction (i.e. the queues are full), then the commitment from the main thread stalls until the verification thread catches up and frees up the necessary space in the buffers.

3. Lifetime-Based Checking Elision (LBCE)

It has been noticed by several researchers that most of the register instances in a datapath are short-lived [34]. We define a value produced by the instruction X to be short-lived (SL) if the architectural register allocated as a destination of X has been redefined (renamed again) before the value generated by X is committed. We further define the instruction that redefines a register allocated to hold a SL value as the redefiner. In our simulations of the SPEC 2000 benchmarks, about 84% of all generated register values were identified as SL. Some statistics pertaining to SL values are presented in Figure 1.

The left set of bars on Figure 1 shows the percentage of SL values for each benchmark. The right set of bars on Figure 1 shows the percentage of generated result values which are SL and there are no unresolved branch instructions between the instruction producing an SL value and its redefiner. We will refer to such results as control-independent short-lived (CISL) values. According to the results of Figure 1, CISL values make up 76.5% of all register results generated by all instructions except loads. We do not account for the load instructions in this statistics because the loads are not verified through the RVQ.

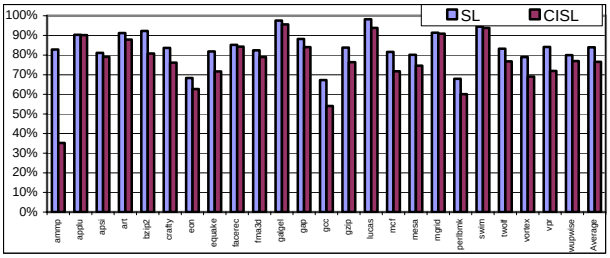

Figure 1: Percentage of register values that are short-lived (SL), and control-independent short-lived (CISL).

The main idea behind our technique of lifetime-based checking elision (hereafter called LBCE) is to avoid the verification of the CISL results and the allocation of the RVQ entries to these non-CISL values. 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. In the rest of the paper, we refer to this technique as Lifetime-Based Checking Elision (LBCE).

To support the capability to recover to a precise and completely verified state following a detection of a transient fault, we rely on the creation of the periodic register and memory state checkpoints, which can be constructed using a two-phase process: checkpoint initiation and checkpoint completion.

Checkpoint initiation. This operation can be associated with any arbitrary instruction and can be performed at any time. For example, assume that the checkpoint initiation is associated with instruction M. The initiation occurs during the cycle when M is inserted into the ROB. At this time, the ROB entries of all older instructions are marked to enforce the verification of all their results, regardless of the decision made by the LBCE algorithm. This essentially creates verification window when all instructions in the ROB are verified without any checking elisions whatsoever.

Checkpoint completion. When the instruction M is verified and no faults are detected, checkpoints of the complete register file and commit rename table are taken. The checkpoint is guaranteed to be complete because all instructions preceding M have committed without problems.

The specific circuit implementation of checkpointing is not central to the LBCE technique. One can either use a separate register file for this purpose (as in [31]), or embed the checkpoint within the register file itself, by backing up each bitcell with a shadow copy [32]. For our evaluations, we assumed the latter alternative.

To buffer a large number of store instructions between two consecutive checkpoints, we use the approach described in [31], which has also been used elsewhere. 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 roll back 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 [33] 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 periodically (e.g. 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 [35] 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.

To understand why the precise state can be created, assume that the instruction M is the youngest dispatched instruction in the ROB at the instant when the decision to create a checkpoint is made. At this moment, the register file can contain some verified (i.e., non-CISL) values, and some non-verified (i.e., CISL) values. However, for every non-verified instance of an architectural register that is stored in the register file, a more recent instance must exist, which was not yet committed and has not been categorized as being subject to verification or not. For example, if the value produced by the instruction writing to architectural register X has not been verified, then there is another in-flight instruction down the stream that writes into the same

architectural register, simply by definition of CISL. Consequently, if we enforce that all the instructions between the ROB_head (which points to the commitment end of the ROB) and instruction M to be unconditionally determined as subject to verification, then at the time of M's verification, a precise state of the register file, with all registers verified, will be available. At this point, the full checkpoint of the register file can be created. Basically, the process of selective verification of values is avoided during the checkpoint creation period, i.e. all values are verified.

We now describe how the two conditions for detecting the CISL values can be implemented. To identify the SL values, we maintain a bit-vector called *Redefined* with one bit for each physical register. When the redefiner is dispatched, it sets the *Redefined* bit corresponding to the previous mapping of its destination architectural register. The *Redefined* bits are reset when the corresponding physical registers are deallocated. Every instruction that has a destination register checks the status of the *Redefined* bit corresponding to its destination physical register one cycle before the commitment. If the bit is set, then the value to be produced is identified as SL.

To detect the absence of unresolved branches between an instruction producing a SL value and its redefiner, each in-flight branch instruction is assigned a unique integer number called a branch tag. We assume that 16 branch instructions could be in-flight at the same time, thus requiring 4 bits to uniquely identify each branch. Each physical register is tagged by the identity of the branch immediately preceding its redefiner. We refer to these bits as *RBT* (Redefiner's Branch Tag) bits. Additionally, a bit-vector called *Unresolved_Branches* is maintained with one bit per branch tag. When a branch tag is allocated, the corresponding bit is set. When a branch correctly resolves, it clears the *UnResolved_Branches* bit corresponding to its branch tag.

The *RBT* bits are set in the following way. When an instruction is renamed, in parallel with setting of the Redefined bit (as explained in the previous paragraph), the instruction also sets the *RBT* bits of the old mapping of its destination register to the id of the most recently dispatched branch. When an instruction is committed, it reads the *RBT* bits associated with its destination physical register and also obtains the branch tag of the oldest in-flight branch. The tag of the oldest branch can be stored in a special 4-bit register and can be dynamically adjusted as the branch instructions are dispatched and committed. Then, using the values of the *RBT* bits, the tag of the oldest branch, and simple shifting logic, a bit mask is created to mask out the tags of the branches located between the instruction in question and its redefiner. The resulting bit mask is then bit-wise ANDed with the *Unresolved_Branches* array, and if the result is a zero, then the instruction is identified as CISL.

If a produced register value is determined to be CISL, then the instruction is simply removed from the datapath and no RVQ entry is established for this instruction (we do not consider load, store or branch instructions here – they are verified through separate structures as described in Section 2). In order to ensure that the verification thread does not attempt to perform the verification for such an instruction, we augment each RVQ entry with a counter which maintains the number of instructions which elided the verification since the previous RVQ entry was established. After the verification thread checks the instruction stored in entry N of the RVQ, it reads out the counter from entry N+1 and skips the verification attempt for the number of consecutive instructions determined by the value of this counter. As in the base case, we read out and compare the values to be verified from the RVQ and the corresponding value from the RF at instruction commitment time. However, we perform dramatically fewer RF reads as we only verify the non short-lived values using the existing read ports on the RF which have a low utilization to begin with. Consequently, the verification overhead resulting from any port contention is much lower compared to the base case which needs to verify all register values.

To accommodate the activities needed to identify CISL properties, the commit process can be pipelined over multiple cycles. To account for these additional delays, we added one extra commit stage to the datapath implementing the LBCE scheme. Finally, it is important to notice that during the checkpoint creation in the LBCE scheme, the progress of the main thread can continue, minimizing the impact on performance.

Table 1: 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
BTB	2048 entry, 2–way set–associative
Branch Predictor	4K entry gShare, 10-bit global history
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 [24] – 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 [13,14]. The SRT model described in Section 2 was implemented in this framework. Table 1 gives the details of the studied processor configuration.

We simulated a total of 24 integer and floating point benchmarks from the SPEC 2000 suite [6], 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 500 million instructions.

For power and energy-related analysis, we implemented full-custom layouts of the RVQ in 0.18m TSMC technology using Cadence design tools. We then performed SPICE simulations using the netlists extracted from these layouts to compute power and energy savings within the RVQ.

5. Results and Discussions

Figure 2 compares the performance of four different redundant multithreaded architectures. 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 three schemes support recovery using post-commit result verification. These are the SRT scheme augmented with the logic to check each and every register value through the RVQ (called SRT+), the DBCE scheme with post-commit verification, and the LBCE scheme proposed in this paper using a checkpointing interval of 100,000 instructions.

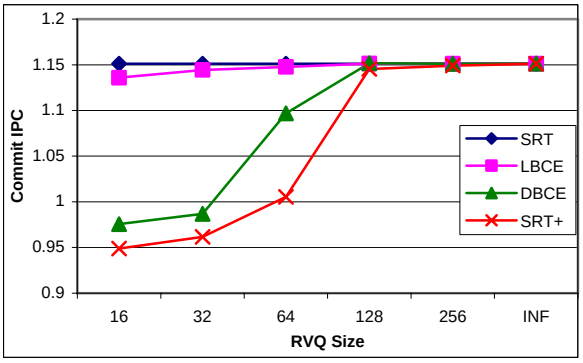

Figure 2: Average commit IPC for various redundant multithreaded architectures for various sizes of the RVQ.

For these experiments, the target slack of 256 instructions (shown to be optimal in [15] and 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 18%, 16.4% and 12.7% 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 15%, 14%, and 4.7% respectively for 16, 32 and 64-entry RVQ compared to the SRT+ design. Finally, the LBCE scheme lowers these percentages further to 1.3%, 0.6% and 0.3%. In other words, 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 3 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 [22] showed 35% for the Spec95 benchmarks). Note that the percent of elisions for LBCE shown here (76.1%) is slightly lower than the percent of CISL values presented in Section 4 (76.5%). This is because the elisions are disabled during the process of checkpoint creation, but that this difference is quite small – only 0.4% on the average. In any case, the LBCE scheme elides a much larger percentage of register value checks, which is manifested in higher IPCs.

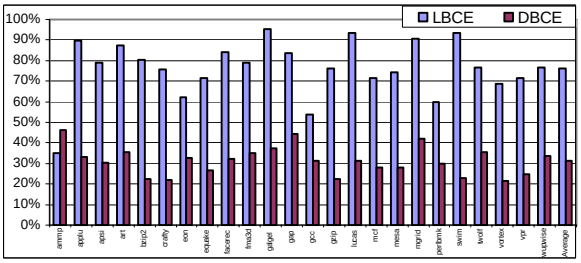

Figure 3: Percentage of register verifications elided using the LBCE and DBCE schemes.

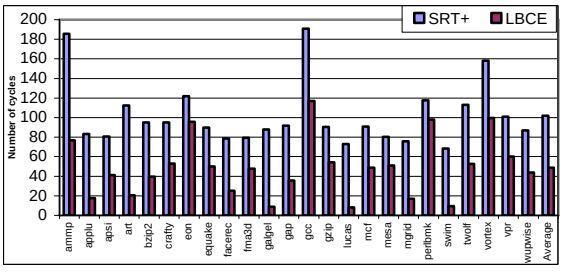

Figure 4: Average number of cycles between allocation and deallocation of an RVQ entry (RVQ lifetime).

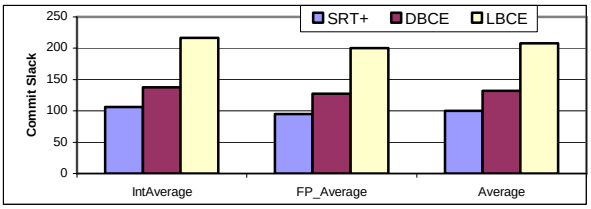

Figure 5: Effective slack length measured in number of instructions at commitment.

Figure 4 presents the average number of cycles between allocation and deallocation of RVQ entries for the SRT+ and LBCE schemes. As shown in the graph, the average lifetime of an RVQ entry is significantly reduced with the LBCE scheme – this value drops from 102 cycles in the SRT+ scheme to 49 cycles with LBCE. This reduction in RVQ entry lifetime is a result of a more efficient RVQ usage by the LBCE scheme.

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 5. 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 2 can be used to understand why that is the case).

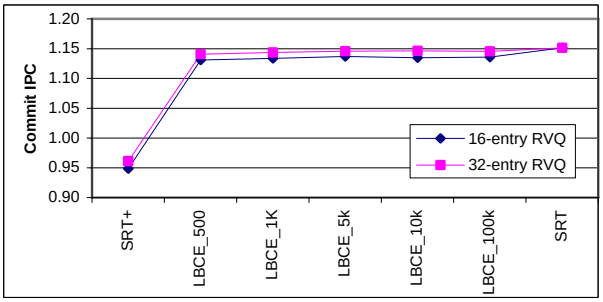

Figure 6: Average commit IPC for the LBCE architecture with various checkpointing frequencies compared to SRT and SRT+.

Next, we evaluate the impact of checkpointing period on the performance of LBCE scheme. Figure 6 presents the commit IPCs for the LBCE technique with various checkpointing intervals compared to the SRT and SRT+ techniques. As seen from the graph, the checkpointing interval has little impact on the effectiveness of the LBCE technique. Specifically, for the machine with a 32-entry RVQ, the difference in IPC between checkpointing interval of 500 instructions and one of 100,000 instructions is only 0.5%. Thus, the checkpointing overhead is relatively small.

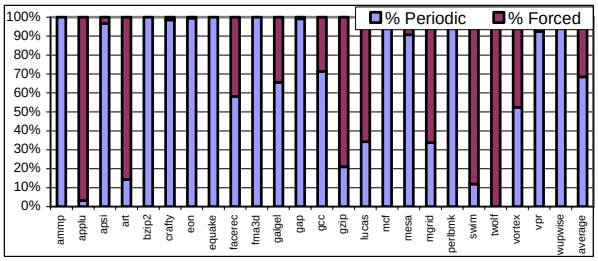

Figure 7: Breakdown of the percentage of checkpoints created periodically versus the percentage of forced checkpoints due to cache behavior.

Recall that there are two triggers for the creation of checkpoints in the LBCE scheme. Checkpoints are created periodically or when required due to the absence of non-volatile data in the cache set for victim selection. Figure 7 presents the data on the percentage of checkpoints created due to each of these triggers. As seen in the graph, 69% of the checkpoints created are induced periodically. The percentage of checkpoints that are created due to all of the sets in a cache line being marked as volatile 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 observed was 2.3%, on *ammp*.

Finally, 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, significantly 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 factors 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. Note also that further power savings *may* be possible in LBCE with little additional performance overhead by reducing the number of ports on the RVQ, as the number of reads required from the RVQ are reduced significantly.

6. Related Work

One approach for concurrent error detection and recovery is to execute two copies of the same program and then compare the results [1, 7, 13, 14, 15, 16, 21, 22]. Ray, Hoe, and Falsafi [14] propose mechanisms for performing such redundant execution within a superscalar processor. Smolens et. al. [20] 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 [15, 16, 22, 7] use the inherent hardware redundancy in SMT and CMP architectures for concurrent error detection. While the SRT scheme described in [15] only aims at detecting transient faults using the SMT support, the follow up study of [22] augments the work of [15] 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 [27], and was extended by CRTR [28] to provide fault recovery using CMPs.

The concept of partial soft error coverage was introduced in [13], 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 [29] employs off-the-shelf microprocessors in lock-step fashion and compares the outputs every cycle. The IBM S/390 [30] uses replicated, lock-stepped pipelines within the processor itself.

7. Concluding Remarks

We presented a technique to reduce the performance and energy overhead of supporting transient fault recovery in a simultaneous multithreaded processor. To this end, we proposed a checkpoint-assisted transient fault recovery mechanism to exploit register lifetime information in the course of verifications and limit the verifications to only the instructions with long lifetimes. Our scheme allows the verifications of 76% of the generated result values to be elided without losing the ability to recover to a precise state. For the same performance and the same effective slack, our technique allows the use of a much smaller verification queue compared to previously proposed design. Alternatively, noticeable performance gains are achieved for the same queue size.

8. Acknowledgements

We would like to thank Aneesh Aggarwal and Deniz Balkan for useful discussions regarding the ideas presented in this paper. We also thank the anonymous reviewers for their valuable comments.

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] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Computer Arch. News*, 1997.

[4] Compaq Computer Corp., "Data integrity for Compaq Non-Stop Himalaya servers," *http://nonstop.compaq.com*, 1999.

[5] G. Hinton, et al, "A 0.18-um CMOS IA-32 Processor With a 4-GHz Integer Execution Unit," *IEEE Journal of Solid-State Circuits*, Vol. 36, No. 11, Nov. 2001.

[6] J. G. Holm, and P. Banerjee, "Low cost concurrent error detection in a VLIW architecture using replicated instructions" *Proc. ICPP-21*, 1992.

[7] M. Gomaa, et. al., "Transient-Fault Recovery for Chip Multiprocessors," *Proc. ISCA-30*, 2003.

[11] S. Mukherjee, et. al., "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," *Proc. Micro-36*, 2003.

[13] M. Gomaa, T.N.Vijaykumar, "Opportunistic Transient Fault Detection", ISCA 2005

[14] J. Ray, J. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," *Proc. Micro-34*, 2001.

[15] S. Reinhardt, and S. Mukherjee, "Transient fault detection via simultaneous multithreading," *Proc. ISCA-27*, June 2000.

[16] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," *Proc. 29th Intl. Symp. On Fault-Tolerant Computing Systems*, 1999.

[18] D. P. Siewiorek and R. S. Swarz, "Reliable Computer Systems Design and Evaluation," *The Digital Press*, 1992.

[19] T. J. Slegel, et al. “IBM’s S/390 G5 microprocessor design,” *IEEE Micro*, 19(2):12-23, March/April 1999.
[20] J.Smolens, et. al., “Efficient Resource sharing in Concurrent error detecting Superscalar microarchitectures ,” *Proc. Micro- 37*, 2004.
[21] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, “Slipstream processors: Improving both performance and fault tolerance,” *In Proc. Micro-33*, December 2000.
[22] T. Vijaykumar, I. Pomeranz, and K. Cheng, “Transient-fault recovery using simultaneous multithreading,” *Proc. ISCA-29*, 2002.
[23] C. Weaver, et. al., “Techniques to Reduce the Soft Error Rate of a High Performance Microprocessor,” *Proc. ISCA-31*, 2004.
[24] J. Sharkey. “M-Sim: A Flexible, Multi-threaded Simulation Environment.” Tech. Report CS-TR-05-DP1, Department of Computer Science, SUNY Binghamton, 2005.
[25] D. Tullsen, et al. “Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor.” in Proc International Symposium on Computer Architecture, 1996.
[26] P. Shivakumar, et al. “Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic”, in Proc DSN, 2002.
[27] S. Mukherjee, et al. “Detailed Design and Evaluation of Redundant Multithreading Alternatives”, in Proc ISCA 2002.
[28] M. Gomaa, et al. “Transient-fault Recovery for Chip Multiprocessors” in Proc ISCA 2003.
[29] Compaq zComputer Corporation, “Data Integrity for Compaq Non-Stop Himalaya Servers”, 1999,
[30] T. Slegel, et al. “IBM’s S/390 G5 Microprocessor Design”, IEEE Micro, 1999.
[31] J. Martinez, et al., “Cherry: Checkpointed Early Resource Recycling in Out-of-Order Processors”, Proc. MICRO 2002.
[32] O.Ergin, et al., “Increasing Processor Performance through Early Register Release”, Proc. ICCD 2004.
[33] M. Kirman, et al., “Cherry-MP: Correctly Integrating Checkpointed Early Resource Recycling in Chip Multiprocessors”, Proc. MICRO 2005.
[34] D. Ponomarev, et al., “Reducing Datapath Energy through the Isolation of Short-Lived Operands”, Proc. PACT 2003.
[35] J. Smolens, et al, “Fingerprinting: Bounding Soft-Error Detection Latency and Bandwidth”, Proc. ASPLOS 2004.