Exploiting Eager Register Release in a Redundantly Multi-Threaded Processor

Niti Madan, Rajeev Balasubramonian
School of Computing, University of Utah
{niti, rajeev}@cs.utah.edu*

Abstract

Due to shrinking transistor sizes and lower supply voltages, transient faults (soft errors) in computer systems are projected to increase by orders of magnitude. Fault detection and recovery can be achieved through redundancy. Redundant multithreading (RMT) is one attractive approach to detect and recover from these errors. However, redundant threads can impose significant performance overheads by competing with the main program for resources such as the register file. In this paper, we propose using eager register release in the main program thread by exploiting the availablity of register values in the trailing thread's register space. This performance optimization can help support a smaller register file and potentially reduce register file access time, power consumption, and increase its immunity towards soft errors.

Keywords: *Reliability, redundant-multithreading, register file design*

1. Introduction

Lower supply voltages and shrinking transistor sizes have led to an exponential increase in soft errors in modern computer systems [21, 28, 36]. Soft errors do not cause any permanent device damage but can result in incorrect program execution for a brief period of time. Soft errors in memory circuits have long been known as a huge concern but immunity in logic circuits has become critical only very recently. Reliability has become a first class design constraint in modern processor design along with power and performance. Several solutions have been proposed to mitigate the effect of these errors at process, circuit, and architecture level. Circuit-level solutions require re-designing of all components and can also add significant design complexity. For this reason, several studies have focused on architectural techniques [2, 8, 13, 18, 22, 23, 24, 25, 26, 30, 34, 35] for fault detection and recovery at modest performance and complexity overheads. Most of these approaches utilize some variant of redundant multithreading (RMT) where a redundant thread executes a copy of the main program for verification either on the same or on separate processor cores. In an era of multi-threaded and multi-core processor technology, it is only natural that these designs be extended to provide reliability, making RMT an attractive choice.

*This work was supported in part by NSF grant CCF-0430063 and by an NSF CAREER award.

The register file is an important resource that determines the size of the in-flight instruction window and hence, instruction-level-parallelism (ILP). In an SMT processor, the register file becomes an even bigger constraint as it has to support logical registers for each thread in addition to the rename registers [33]. Improving the efficiency of register allocation will lead to performance improvements, especially for multi-threaded workloads. It can also allow a processor to match a baseline system's throughput with a smaller register file, potentially causing improvements in power consumption, clock speed, temperature, and potentially freeing up area to implement ECC/parity. The register file is already a vulnerable structure for single event upsets [11, 35] and increasing rates of multi-bit upsets [15, 31] will require more aggressive ECC/parity schemes.

In certain RMT implementations, the redundant thread (also known as trailing thread) co-executes on a single SMT core with the main program thread (also known as leading thread). Just as in any multi-threaded system, this exerts pressure on shared processor resources such as the register file and issue queue. To address this drawback of RMT, recent papers have attempted to improve register file efficiency in such processors. Abu-Ghazaleh et al. [1] avoid allocating registers for transient short-lived values. Kumar and Aggarwal [12] employ the following optimizations: (i) two narrow operands share a single register, (ii) two identical register values share the same physical register. For processors without RMT, in order to improve the register file's resiliency to soft errors, Hu et al. [9] and Ergin et al. [7] propose that a single register can store copies of a narrow operand, while Memik et al. [17] use dead or free registers to opportunistically create copies of register values.

In this paper, we propose a novel register allocation mechanism that takes advantage of redundancy within an RMT processor. In a traditional register file system, the older mapping of a logical register is de-allocated only when the overwriting instruction commits. This guarantees that if the overwriting instruction gets squashed due to a misprediction, then the older mapping can be used for reinstating the architectural state. If a copy of the older mapping exists outside the physical register file, the older mapping can be de-allocated early. By exploiting the availability of register values in the trailing thread's register space, we can employ eager release in the leading thread's register file. This optimization can boost the leading thread's performance while allowing a very small number of errors to go un-detected. We quantify these effects for a number of RMT processor models.

It must be noted that eager register release can yield performance benefits in any processor model, not just in RMT implementations. It is especially well suited to RMT implementations because (i) copies of register values already exist in the system, and (ii) RMT implementations are typically multi-threaded and are more prone to register file bottlenecks.

The paper has been organized as follows. Section 2 describes the redundant multi-threading implementations that serve as baseline processor models in this study. Section 3 describes the eager register release mechanism for the leading thread. The proposed ideas are evaluated in Section 4 and we contrast our approach with related work in Section 5. Section 6 summarizes the conclusions of this study.

2. Baseline Reliable Processor Models

We first discuss design aspects that are common to all the baseline RMT implementations studied in this paper. The leading thread executes ahead of its counterpart trailing thread by a certain amount of slack to enable checking for errors. The leading thread communicates its committed register results to the trailing thread for comparison of values to detect faults. Load values are also passed to the trailing core so it can avoid reading values from memory that may have been recently updated by other devices. Thus, the trailing thread never accesses the L1 data cache. This implementation uses *asymmetric commit* to hide intercore communication latency (if leading and trailing threads execute on separate cores) – the leading thread is allowed to commit instructions before checking. The leading core commits stores to a store buffer (StB) instead of to memory. The trailing core commits instructions only after checking for errors. This ensures that the trailing core's state can be used for a recovery operation if an error occurs. The trailing core communicates its store values to the leading core's StB and the StB commits stores to memory after checking. We have used asymmetric commit even when leading and trailing threads execute in SMT fashion on the same core. This enables reduced design complexity and improves performance as the trailing thread need not verify the leader's speculative register values.

To facilitate communication of values between leading and trailing threads, first-in-first-out register value queues (RVQ) and load value queues (LVQ) are used. As a performance optimization, the leading core also communicates its branch outcomes to the trailing core (through a branch outcome queue (BOQ)), allowing it to have perfect branch prediction. If the slack between the two threads is at least as large as the re-order buffer (ROB) size of the trailing thread, it is guaranteed that a load instruction in the trailing thread will always find its load value in the LVQ. When external interrupts or exceptions are raised, the leading thread must wait for the trailing thread to catch up before servicing the interrupt. The ICOUNT fetch policy [32] is used to determine priority for co-scheduled threads on an SMT processor as long as the slack value is within an acceptable range.

The assumed fault model is exactly the same as in [8, 22]. The following condition is required in order to detect a single fault:

- The data cache, LVQ, and buses that carry load values must be ECC-protected as the trailing thread directly uses these load values.

Other structures in each core (including the RVQ) need not have ECC or other forms of protection as disagreements will be detected during the checking process. The BOQ need not be protected as long as its values are only treated as branch prediction hints and confirmed by the trailing pipeline.

The following additional condition is required in order to detect and recover from a single fault:

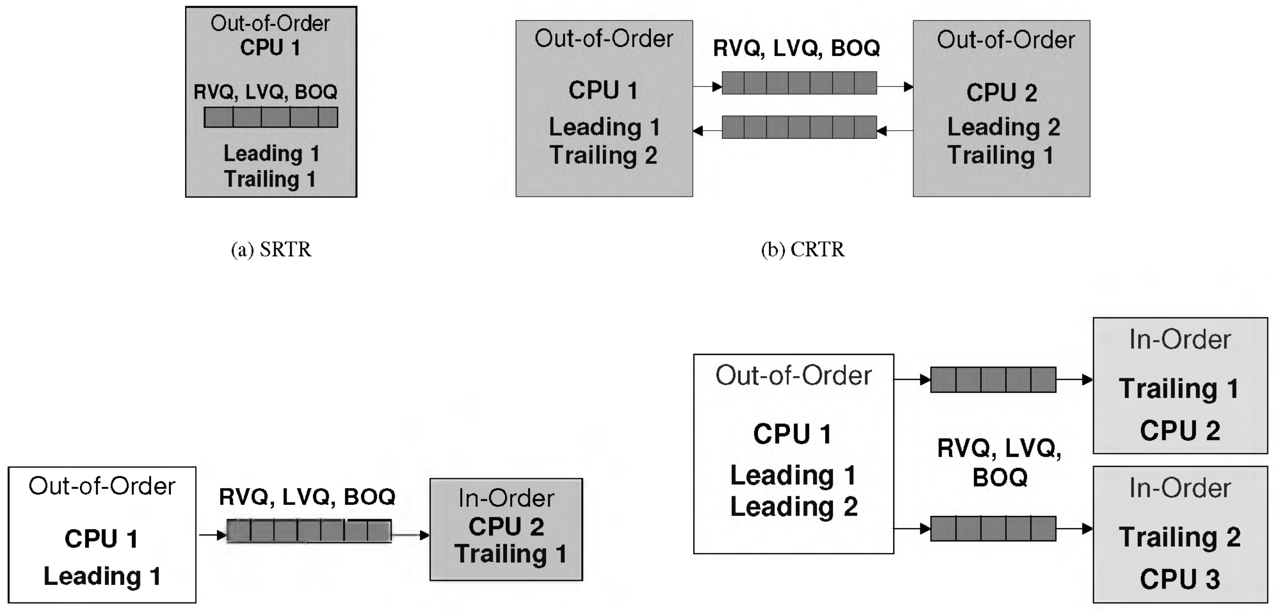
- When an error is detected, the register file state of the trailing thread is used to initiate recovery. The trailing thread's register file must be ECC-protected to ensure that values do not get corrupted once they have been checked and written into the trailer's register file.

Similar to the baseline model in [8, 22], we assume that the above condition is not met (*i.e.*, the trailer's register file is not ECC-protected). Hence, a single fault in the trailer's register file can only be detected [1]. All other faults can be detected and recovered from.

We consider RMT implementations that make various design choices along the following axes: (i) the cores that leading and trailing threads execute on, (ii) power-efficient execution of trailing threads when possible, and (iii) support for single and multi-thread workloads.

- **Simultaneously and Redundantly Threaded processor with Recovery (SRTR)** is based on the fault-tolerant processor model proposed in [34]. In this implementation, the leading and the trailing thread co-execute in SMT fashion on the same core as shown in Figure 1(a). We have changed the model proposed in SRTR and added asymmetric commit to it. The hardware cost of redundancy is relatively low in SRTR, making it an attractive solution. However, SRTR suffers from significant performance overheads as the trailing thread puts pressure on shared resources.

[1] If no ECC is provided within the register fi le, Triple Modular Redundancy will be required to detect and recover from a single fault.



(a) SRTR (b) CRTR

(c) ST-P-CRTR (d) MT-P-CRTR

Figure 1. **RMT Design Space**

- **Chip-level Redundantly-Threaded processor with Recovery (CRTR)** is the model proposed by [8, 22] where each core is a dual-threaded SMT processor. In the CRTR architecture, the trailing thread of one application shares its core with the leading thread of a different application (shown in Figure 1(b). This architecture has been optimized for higher throughput for multi-threaded workloads.
- **Single-thread Power-efficient Chip-level Redundantly-Threaded processor with Recovery (ST-P-CRTR)** is a power-efficient implementation of a single-thread RMT processor [13, 14]. It extends some of DIVA's [2] concepts to general-purpose cores. This model executes the leading thread on an aggressive out-of-order processor and its trailing thread on a simple in-order core as shown in Figure 1(c). An in-order core by itself is not capable of matching the leading thread's throughput even with perfect caching and branch prediction. The RVQ is therefore also made to carry instruction source operands to enable perfect value prediction at the trailing core. This optimization does not compromise fault coverage because the source operands are also verified at the trailer. Even though this increases inter-core bandwidth requirements, the net effect is a reduction in power consumption because of the efficiency of a low-frequency in-order checker core.
- **MT-P-CRTR** is a multi-threaded extension of ST-P-CRTR. Two different leading threads execute in SMT fashion on a single out-of-order core, while their corresponding trailing threads execute on separate simple in-order cores [13, 14] as shown in Figure 1(d). CRTR has higher performance than MT-P-CRTR. This is because each leading thread in CRTR is co-scheduled with a trailing thread that does not execute speculative instructions and therefore poses less contention for the SMT core's resources. MT-P-CRTR consumes much lower power than CRTR. Both, ST-P-CRTR and MT-P-CRTR, employ dynamic frequency scaling (DFS) to allow the trailing core to match the leading core's throughput, while consuming the least possible power.

In this paper, we are not necessarily arguing for one baseline implementation over the other. These models represent different points on the performance-power-complexity curves and enable a more comprehensive evaluation of our register allocation techniques.

3. Proposed Register Allocation Policies

It is a well-known result that register utilization in modern out-of-order processors is extremely inefficient. Registers are allocated long before a result is actually written into them and de-allocated much after their last use. Many papers have targeted both of the inefficiencies above (for example, [3, 5, 19]). In a conventional processor, physical registers are conservatively de-allocated. When a logical (architectural) register is over-written by a new instruction, it is assigned a new physical register. However, the old register mapping for that logical register cannot be de-allocated immediately. If the new instruction is squashed (because of a branch mis-speculation or exception), the old mapping has to be re-instated. Therefore, the old mapping can be freed only when the new instruction commits. It has been proposed [3] that old mappings be copied away into a larger second-level register file that is off the critical path. This allows registers in the first-level register file to be recycled sooner, improving their utilization and supporting a larger in-flight instruction window. Since old mappings are not discarded, the processor can still recover from mis-speculations and exceptions.

Such a technique is especially well-suited to an RMT processor. The RMT processor already maintains multiple copies of register values. Hence, threads can quickly recycle registers and be guaranteed of safe recovery in case it is warranted. The only catch is that recovery may not be possible if a redundant copy is already corrupted. The RMT implementation is therefore no longer capable of detecting every single fault. Since most systems do not expect zero FIT rates, this may represent an acceptable performance-reliability trade-off.

3.1. Implementation Details

We first describe our eager register release mechanism in the context of the single-thread ST-P-CRTR model, where leading and trailing threads execute on separate cores (Figure 1(c)). This basic design can be easily extended to other RMT implementations.

In a baseline system, a physical register is de-allocated when the instruction that over-writes the corresponding logical register is committed. In the proposed system with eager register release, a physical register *P* belonging to a leading thread is de-allocated when the following conditions are met:

- the physical register value has been read by all consuming instructions in the pipeline,
- the instruction that writes to the physical register (*instr A*) has been committed by the leading thread and the physical register value has been copied into the RVQ,
- a new instruction that over-writes the corresponding logical register has entered the pipeline (*instr B*).

We chose to not implement a more aggressive release policy in order to minimize the number of recoveries.

After being released eagerly, physical register *P* can be assigned to a new instruction. If a branch mis-predict or exception occurs between *instr A* and *instr B*, physical register *P* is freed because all instructions after the branch are squashed. The old value of physical register *P* must now be re-instated. If the trailing thread has not executed *instr A*, the value of register *P* will be found in the RVQ. The leader re-instates this value into physical register *P* and continues. Even if the result in the RVQ is corrupted, it will undergo a check at the trailer and the error will be flagged. If the trailing thread has already executed *instr A*, the checked value will be found in the trailing thread's register file (in register *Q*). Since the leading thread has not committed *instr B*, the trailing thread would also not have executed *instr B* (note that the trailer maintains a minimum slack). Hence, the register value *Q* would not have been over-written and will represent the most up-to-date mapping of the corresponding logical register in the trailer. The value in register *Q* is copied into register *P* and the leading thread resumes execution.

In the second case above, it is possible that an error may go un-detected. When the trailing thread commits *instr A*, it verifies the result in the RVQ before storing it into register *Q*. If the value in *Q* then gets corrupted, the leading thread, on recovery, will also adopt the incorrect value. The error will never be detected as both threads will continue to agree on all results. In a baseline RMT system, if *Q* does get corrupted, an error will be flagged because the consumer of *P* in the leader and the consumer of *Q* in the trailer will eventually dis-agree. However, the probability of an un-detected error in the new system is extremely low. The following conditions must be met: (i) the leader has not committed *instr B*, (ii) the trailer has committed *instr A*, and (iii) the specific register *Q* is corrupted. Since we maintain a large slack and since successive writers to a logical register are not greatly separated, the likelihood that (i) and (ii) are both true is extremely small (quantified in the next section).

We now briefly examine the storage and control structures required to implement the above copy and recovery operations. Each physical register in the leading core requires a bit to track whether the corresponding logical register has been over-written (*overwrite bit*) and another bit to track if the value has been copied into the RVQ (*in_RVQ bit*). If the register value has been copied into the RVQ, then the corresponding RVQ address also needs to be stored (*RVQ_address* field). Each physical register maintains a counter for the number of outstanding consumers (*pending_consumers*). This counter is incremented when consumers are dispatched and decremented when consumers leave the issue queue. Each physical register also keeps track of the instruction that it is assigned to. A *usage table* structure does the above book-keeping for each physical register. The ROB entry of the over-writing instruction (*instr B* in the example above) keeps track of the recovery operations it must initiate if it is squashed. This includes maintaining the instruction number *inum* (*instr A* in the example above), logical register ID *lreg* for the physical register (*P*) that it caused to be released eagerly and a *de-allocate bit*.

Figure 3 shows a block level depiction of our eager release implementation. Our technique can release a physical register eagerly any time after the previously described conditions are met. In our simulations, we check the usage table every cycle to determine the registers that can be released eagerly. An alternative approach, not considered in

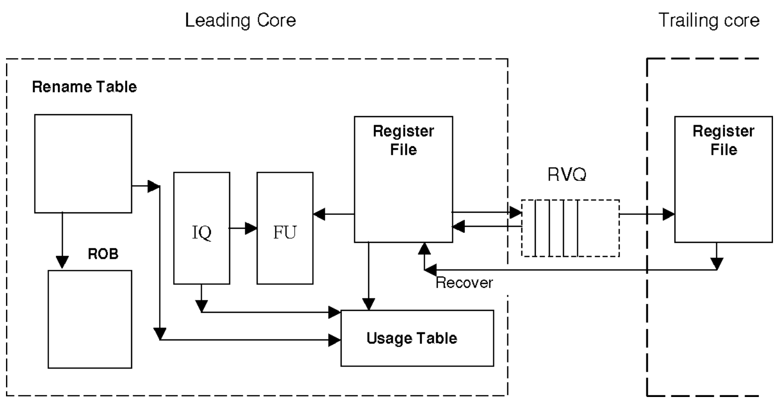


Figure 2. **Block-level Implementation of Eager Release in a ST-P-CRTR model**

this study, can release physical registers eagerly when the number of registers in the free pool falls below a certain threshold.

These auxiliary structures and control logic can impose a non-trivial complexity overhead. However, this overhead may be justifiable as it enables significantly higher throughput or the design of a small register file. SMT workloads can especially benefit from eager release and potentially tolerate the complexity overhead for these structures. For example, the register file may now be small enough that it can be implemented as a single-cycle structure or it has the latency/power/area budget to implement ECC.

Since the auxiliary structures are outside the sphere of replication, they are also vulnerable to soft errors. Luckily, faults in these structures do not result in silent data corruption (SDC). Consider an example where a bit in the usage table is affected and the *pending_consumers* field reaches zero even though active consumers exist in the pipeline. If the corresponding register is released eagerly and re-allocated, the pending consumer may read an incorrect value and produce a wrong result. Such an error will be detected by the redundant instruction in the trailing thread. This argument holds true even when other fields in the usage table are corrupted by soft errors.

For most of this paper, the eager register release mechanism is only applied to physical registers in the leading thread. The technique may also apply to the trailing thread. If a register value is eagerly discarded by the trailing thread, it may be unable to recover from a branch mis-predict. Note that a branch mis-predict in the trailer happens only when a soft error manifests. A missing result in the trailing register file can therefore hamper recovery. Secondly, an ILP improvement in the trailer may be beneficial because the trailer can further scale down its frequency and save power. We evaluated this technique, but found that the ILP improvement of the trailing thread was marginal in most cases because high-ILP threads are already efficient at re-cycling registers. Because of the above two reasons, we only consider eager register release for leading threads.

Branch Predictor	Comb. of bimodal,2-level (per core)
Level 1 and 2 Predictor	16384 entries
Branch Mpred Latency	12 cycles
Instruction Fetch Queue	32 (per Core)
Fetch/Dispatch/Commit width	4 (fetch upto 2 branches)
IssueQ size	40 (Int) 30 (FP) (per Core)
Reorder Buffer Size	160 (per Thread)
LSQ size	200 (per Core)
(Single thread) L1 D,I-cache	32KB 2-way (per Core)
(Multi-thread) L1 D,I-cache	128KB 2-way (per Core)
L2 unifi ed cache	2MB 8-way, 20 cycles (per Core)
Memory Latency	300 cycles for the fi rst chunk
RVQ/BoQ/LVQ sizes	600/200/400 entries

Table 1. **Simplescalar Simulation Parameters**

4. Results

4.1. Performance Evaluation

We use a multi-threaded version of Simplescalar-3.0 [4] for the Alpha AXP ISA for our simulations. The simulator has been extended to implement both homogeneous and heterogeneous CMP architectures. We have modeled each core as a 2-way SMT or as an in-order processor. Table 1 shows relevant simulation parameters. CACTI-3.2 [27] has been used to compute area, performance, and power results for different register file configurations.

As an evaluation workload, we use the 8 integer and 8 floating point benchmark programs from the SPEC2k suite that are compatible with our simulator. The executables were generated with peak optimization flags. The programs were fast-forwarded for 2 billion instructions, executed for 1 million instructions to warm up various structures, and measurements were taken for the next 100 million instructions. To evaluate multi-threaded models, we have formed a benchmark set consisting of 10 different pairs of programs. Programs were paired to generate a good mix of high IPC, low IPC, FP, and Integer workloads. Table 2 shows our benchmark pairs. Multithreaded workloads are executed until the first thread commits 100 million instructions.

Benchmark Set	Set #	IPC Pairing	Benchmark Set	Set #	IPC Pairing
art-applu	1	FP/FP/Low/High	bzip-fma3d	2	Int/FP/Low/High
bzip-vortex	3	Int/Int/Low/Low	eon-art	4	Int/FP/High/Low
eon-vpr	5	Int/Int/High/High	gzip-mgrid	6	Int/FP/Low/Low
mesa-equake	7	FP/FP/High/High	swim-lucas	8	FP/FP/Low/Low
twolf-equake	9	Int/FP/High/High	vpr-gzip	10	Int/Int/High/Low

Table 2. Benchmark pairs for the multi-threaded workload.

4.2. Performance Evaluation

For all our experiments, we set the ROB size to 160 per thread and attempt to fill the window with a much smaller set of registers. We evaluate the effect of the baseline conventional register de-allocation policy as well as the eager register release policy as the register file size is gradually increased. We initially assume that there is no performance penalty for recovery of eagerly released register values, *i.e.*, this recovery happens in parallel with the fetch of instructions from the correct branch path. Later, we show the effect of non-zero recovery latencies.

Figure 3 shows the IPC curve for the SRTR model where both leading and trailing threads execute on the same SMT core. With eager register release, a physical register file of 100 entries has performance equivalent to a base model with 160 register entries. Compared to a baseline processor with a 100-entry register file, the eager release policy enables a 10% performance improvement. For the CRTR model (Figure 4), we see a similar result trend as SRTR, where the eager release policy can match the baseline's performance with 37.5% fewer register entries. With a fixed register file size of 100, the eager release policy yields a 34% performance improvement over the baseline policy. A similar result is also seen for the MT-P-CRTR model (Figure 6) where the allocation of threads to cores is different and the trailing threads are frequency-scaled. Finally, we verify our results for the single-thread ST-P-CRTR model (Figure 5). Since only a single thread context executes on each core in this case, we find that a 50-entry register file with eager release is equivalent in performance to an 80-entry conventional register file. Floating-point programs exhibit higher performance improvements with eager register release – for a 50-entry register file, eager release causes an improvement greater than 20% for five FP programs (*swim, art, mesa, mgrid, lucas*) and one integer program (*vpr*). Programs with poor branch prediction accuracies do not benefit as much from quick register re-cycling and larger in-flight windows – *gcc, equake, eon,* and *fma3d* show an improvement of less than 3%. In general, we find that models that execute two leading threads on the same core (MT-P-CRTR) benefited much more from eager register release because of the much higher pressure on the register file.

We observe that the cost of copying eager released values back to the leading thread on a recovery is not very high. For a single-thread 100M instruction simulation, about 70 million registers are released eagerly, of which only 6% are copied back as part of a branch mispredict recovery. The

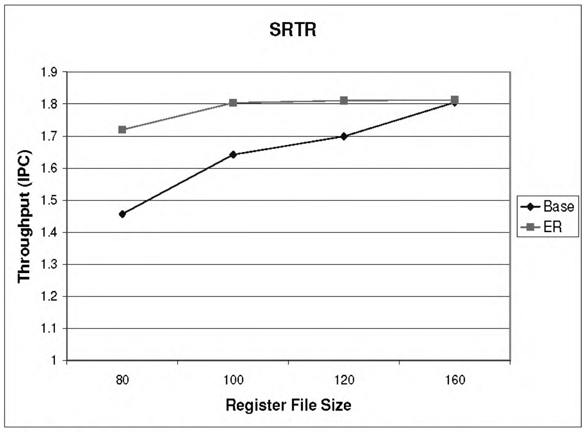


Figure 3. IPCs for different register file sizes with and without eager release for SRTR

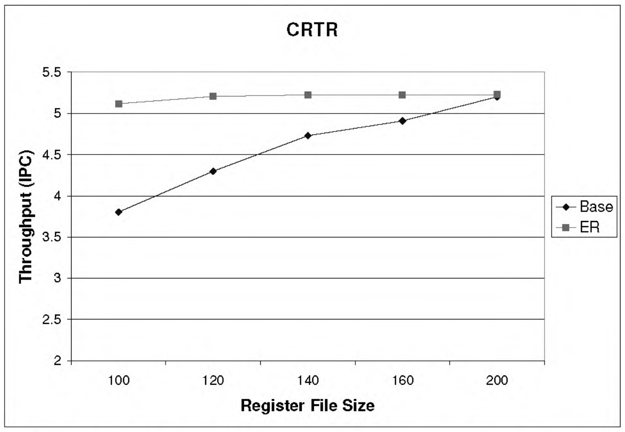


Figure 4. IPCs for different register file sizes with and without eager release for CRTR

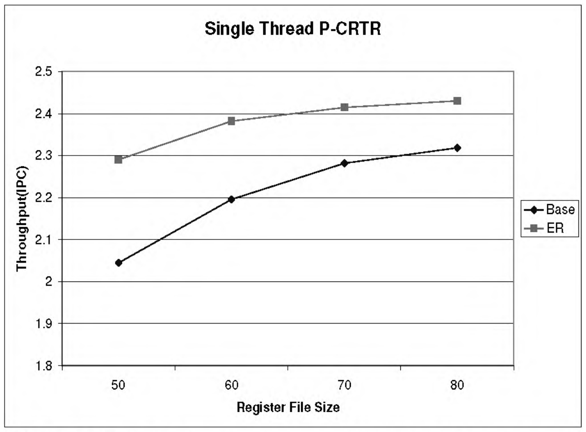


Figure 5. IPCs for different register file sizes with and without eager release for ST-P-CRTR

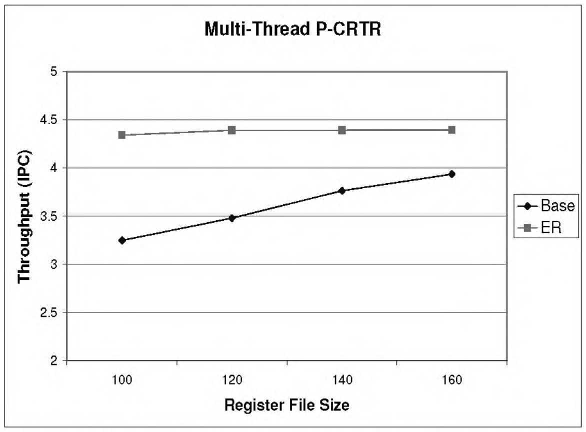


Figure 6. **IPCs for different register file sizes with and without eager release for MT-P-CRTR**

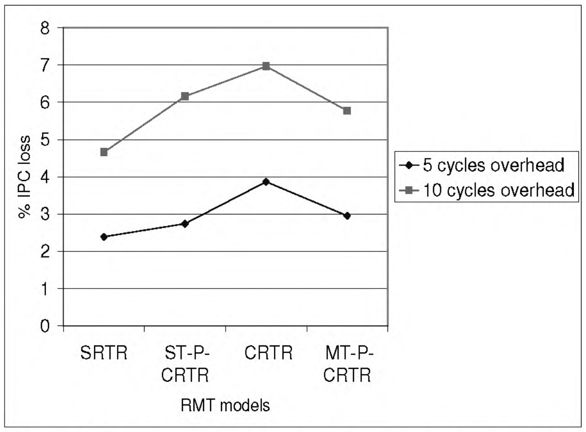


Figure 7. **IPC loss due to register value copying overhead in all RMT models**

Single Thread Register File			
RF Size	**Energy (nJ)**	**Access time (ns)**	**Area (cm2)**
50	0.78	0.506	0.00764
60	0.805	0.527	0.00821
70	0.848	0.59	0.00958
80	0.879	0.627	0.0101
Multi-thread Register File			
100	2.41	1.005	0.0421
120	2.617	1.145	0.0466
140	2.816	1.248	0.051
160	3.027	1.411	0.0555
200	3.432	1.72	0.0644

Table 3. **Access time, energy and area results derived from CACTI (90nm technology)**

programs *bzip* and *eon* have the highest percentage (13%) of copy-backs, while programs such as *swim, lucas*, and *mgrid* have a low branch mispredict rate and much fewer total copy-backs. The cost of copying results back depends on the program's branch prediction rate. On average, each branch mispredict requires that 6.6 register values be copied back to the leading thread. We expect that the cost of copying results back into the leader can be hidden by the cost to fill the pipeline with correct-path instructions. As a sensitivity study, Figure 7 shows the average IPC loss for each RMT implementation if the latency for a branch mispredict recovery is increased by 5 and 10 cycles. For a 5-cycle penalty, the maximum observed performance degradation is 4%. In all our RMT implementations, the trailing thread continues to execute while the leading thread recovers from a branch mis-predict. If we pessimistically assume that branch recovery consumes all register ports and even trailing threads are stalled during the recovery phase, the additional loss in performance is less than 1% for all models.

4.3. Fault Injection Analysis

As described in Section 3, the eager release mechanism can lead to un-detected faults. This can happen if the gap between successive writes to a logical register is greater than the slack between leading and trailing threads. We computed the intervals between successive writes to a logical register and observed that 90% of the time, this interval was less than 100 instructions. For most of our simulations, the average slack hovers around 500 instructions. This ensures that for more than 99% of all cases, an eagerly released register value can be found in the RVQ, not in the trailer's register file. We evaluated the effect on error coverage by injecting faults into our Simplescalar simulations[2]. Once every 1000 cycles, a valid bit is flipped in a random register in the trailer. Only 0.0004% of all these incorrect values were copied back into the leader, causing an error to go un-detected. This analysis is conservative because some errors will get architecturally masked and not lead to silent data corruption (SDC).

4.4. Discussion

Our results so far have shown that eager register release causes a minor decrease in fault coverage, but can improve a fixed register file's performance by up to 34%. Alternatively stated, a 50-entry register file with eager release, can match the performance of an 80-entry register file. We modified CACTI-3.2 to model access time, area, and energy of various register file organizations at 90nm technology. The move from an 80-entry register file to a 50-entry register file has a number of favorable implications, quantified below.

[2]Not all faults at the transistor level manifest themselves at the microarchitectural level. The use of a functional simulator such as Simplescalar allows us to carry out our analysis only for faults that manifest themselves in data and control paths at the microarchitecture level modeled by Simplescalar.

Table 3 shows CACTI results for both single- and multi-threaded register files. We assume that a single-threaded register file has 8 read ports and 4 write ports. The multi-threaded register file has 16 read and 8 write ports. If the register file is a cycle-time constraint, a 50-entry register file can enable a 19% increase in clock speed, compared to the 80-entry register file. The 50-entry register file also consumes 11% less energy and 25% less area. Similar observations are made when the multi-threaded register file is shrunk from 160 to 100 entries. It has been reported that an ECC implementation imposes a 6% power and 16% area overhead on the register file. By implementing a smaller register file, we may have the power and area budget within the register file to implement ECC [15, 31]. The computed ECC overheads are for a SEC-DED (single error correction and double error detection) ECC scheme. Multi-bit errors will require even more aggressive ECC/parity protection schemes such as DEC-TED (double error correction and triple error detection). Note that a baseline RMT implementation can guarantee error detection and recovery only if the trailing register file has ECC protection. Hence, the ability to implement ECC in the register file has important implications for error recovery.

In this preliminary study, we have not quantitatively compared the benefits of the eager release mechanism with other recent register file proposals, such as those that exploit narrow-width operands (for example, [12]). We feel that the eager-release strategy is orthogonal to narrow-width optimizations as the two techniques target different sources of register file inefficiency. We therefore expect that the two techniques can be combined to yield significantly greater speedups. The implementations of either technique entail non-trivial complexity and will likely determine the commercial feasibility of each approach.

5. Related Work

Many fault-tolerant architectures [2, 8, 22, 25, 26, 29, 34] have been proposed over the last few years. AR-SMT [26] was the first design to use multi-threading for fault detection. Mukherjee et al. proposed fault detection using simultaneous multi-threading and chip-level redundant multi-threading [22, 25]. Vijaykumar et al. augmented the above techniques with recovery mechanisms [8, 34]. Some designs such as DIVA [2] use an in-order checker core to verify the results of an aggressive superscalar processor.

Smolens et al. study the performance impact of redundant execution on the issue logic and ROB [30]. Recently, many researchers have looked into efficient techniques to improve register file efficiency and reliability [1, 7, 9, 12, 16, 17]. Memik et al. [17] propose utilizing free registers and predicted dead registers to store register value copies for increasing the register file's immunity to soft errors. Memik et al. also present a reliability model that computes the probability of soft error occurrence as a function of the operating clock frequency [16]. They propose that a register file can be overclocked for performance improvement and the resulting increase in soft error rate can be mitigated by employing their earlier technique [17]. In [9, 12], narrow-width operands are allocated a single register to reduce register file resource redundancy. Kumar and Aggarwal [12] apply register re-use and narrow-width operand register sharing techniques to reduce the performance and power overheads in simultaneous redundant multithreading. Similarly, Hu et al. [9] eliminate the requirement of copy registers by storing 2 copies of a narrow-width operand in a single register.

A number of implementations for early register release in non-RMT superscalars have been proposed in recent years [3, 6, 10, 20]. Continued interest in this area may well produce a complexity-effective implementation in the near future. Ergin et al. [6] introduce checkpointed register files to implement early register release. Jones et al. [10] use a compiler-assisted early register release technique. Balasubramonian et al. [3] propose using a two-level register file where the first level register file is smaller in size and eagerly released registers are stored in a second level register file. Our proposal is the first application of the eager release technique to an RMT processor.

6. Conclusions

In this paper, we have shown that redundant copies of a register in an RMT system make it a perfect candidate for the eager register release policy. The quick re-cycling of registers allows the register file to support a much larger window of in-flight instructions. The impact on fault coverage is marginal. We show that this technique is effective for a number of RMT implementations, with multi-threaded throughput being improved by up to 34%. We also show that a 100-entry register file can match the throughput of a 160-entry register file. A smaller register file has favorable implications on clock speed, power, area, and even reliability by making ECC more affordable. For future work, we plan to investigate more complexity-effective implementations of eager register release.

References

[1] N. Abu-Ghazaleh, J. Sharkey, D. Ponomarev, and K. Ghose. Exploiting Short-Lived Values for Low-Overhead Transient Fault Recovery. In *Proceedings of Workshop on Architectural Support for Gigascale Integration*, June 2006.

[2] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of MICRO-32*, November 1999.

[3] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the Complexity of the Register File in Dynamic Superscalar Processors. In *Proceedings of MICRO-34*, pages 237–248, December 2001.

[4] D. Burger and T. Austin. The Simplescalar Toolset, Version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[5] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-Banked Register File Architectures. In *Proceedings of ISCA-27*, pages 316–325, June 2000.

[6] O. Ergin, D. Balkan, D. V. Ponomarev, and K. Ghose. Increasing processor performance through early register release. In *ICCD*, 2004.

[7] O. Ergin, O. Unsal, X. Vera, and A. Gonzalez. Exploiting Narrow Values for Soft Error Tolerance. *Computer Architecture Letters*, 5, June 2006.

[8] M. Gomaa, C. Scarbrough, and T. Vijaykumar. Transient-Fault Recovery for Chip Multiprocessors. In *Proceedings of ISCA-30*, June 2003.

[9] J. Hu, S. Wang, and S. Ziavras. In-Register Duplication: Exploiting Narrow-Width Value for Improving Register File Reliability. In *DSN*, 2006.

[10] T. M. Jones, M. F. P. O'Boyle, J. Abella, A. Gonzalez, and O. Ergin. Compiler directed early register release. In *IEEE PACT*, 2005.

[11] T. Karnik, P. Hazucha, and J. Patel. Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes. In *IEEE Transactions on Dependable and Secure Computing*, volume 01, pages 128–143, 2004.

[12] S. Kumar and A. Aggarwal. Reduced Resource Redundancy for Concurrent Error Detection Techniques in High Performance Microprocessors. In *Proceedings of HPCA-12*, Feb 2006.

[13] N. Madan and R. Balasubramonian. Power-Effi cient Approaches to Reliability. Technical Report UUCS-05-010, University of Utah, December 2005.

[14] N. Madan and R. Balasubramonian. A First-Order Analysis of Power Overheads of Redundant Multi-Threading. In *Proceedings of the Second Workshop on the System Effects of Logic Soft Errors (SELSE-2)*, April 2006.

[15] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong. Characterization of multi-bit soft error events in advanced SRAMs. In *Digest of International Electron Devices Meeting*, 2003.

[16] G. Memik, M. H. Chowdhury, A. Mallik, and Y. I. Ismail. Engineering over-clocking: Reliability-performance tradeoffs for high-performance register fi les. In *DSN*, pages 770–779, 2005.

[17] G. Memik, M. Kandemir, and O. Ozturk. Increasing Register File Immunity to Transient Errors. In *Proceedings of DATE-2005*, Mar 2005.

[18] A. Mendelson and N. Suri. Designing High-Performance and Reliable Superscalar Architectures: The Out-of-Order Reliable Superscalar O3RS Approach. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2000.

[19] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, and V. Vinals. Delaying Physical Register Allocation through Virtual-Physical Registers. In *Proceedings of MICRO-32*, pages 186–192, November 1999.

[20] T. Monreal, V. Vinals, A. Gonzalez, and M. Valero. Hardware schemes for early register release. In *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing (ICPP'02)*, 2002.

[21] S. Mukherjee, J. Emer, and S. Reinhardt. The soft-error problem: An architectural perspective. In *Proc. of 11th International Symposium on High Performance Computer Architecture HPCA*, 2005.

[22] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed Design and Implementation of Redundant Multithreading Alternatives. In *Proceedings of ISCA-29*, May 2002.

[23] M. Rashid, E. Tan, M. Huang, and D. Albonesi. Exploiting Coarse-Grain Verifi cation Parallelism for Power-Effi cient Fault Tolerance. In *Proceedings of PACT-14*, 2005.

[24] J. Ray, J. Hoe, and B. Falsafi . Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery. In *Proceedings of MICRO-34*, December 2001.

[25] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of ISCA-27*, pages 25–36, June 2000.

[26] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of 29th International Symposium on Fault-Tolerant Computing*, June 1999.

[27] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Technical Report TN-2001/2, Compaq Western Research Laboratory, August 2001.

[28] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinatorial Logic. In *Proceedings of DSN*, June 2002.

[29] T. J. Slegel, R. M. A. III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 Microprocessor Design. In *IEEE Micro*, volume 19, pages 12–23, 1999.

[30] J. Smolens, J. Kim, J. Hoe, and B. Falsafi . Effi cient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures. In *Proceedings of MICRO-37*, December 2004.

[31] M. Spica and T. Mak. Do We Need Anything More Than Single Bit Error Correction (ECC)? In *Proceedings of International Workshop on Memory Technology, Design and Testing (MTDT'04)*, 2004.

[32] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of ISCA-23*, May 1996.

[33] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of ISCA-22*, pages 392–403, June 1995.

[34] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery via Simultaneous Multithreading. In *Proceedings of ISCA-29*, May 2002.

[35] N. Wang, J. Quek, T. Rafacz, and S. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *Proceedings of DSN*, June 2004.

[36] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt. Techniques to reduce to soft-error rate in high performance microprocessors. In *Proc. of 31st Annual International Symposium on Computer Architecture ISCA*, 2004.