Dynamic Transient Fault Detection and Recovery for Embedded Processor Datapaths

Garou Bournoutian
University of California, San Diego
9500 Gilman Dr. #0404
La Jolla, CA 92093-0404
garou@cs.ucsd.edu

Alex Orailoglu
University of California, San Diego
9500 Gilman Dr. #0404
La Jolla, CA 92093-0404
alex@cs.ucsd.edu

ABSTRACT

As microprocessors continue to evolve and grow in functionality, the use of smaller nanometer technology scaling coupled with high clock frequencies and exponentially increasing transistor counts dramatically increases the susceptibility of transient faults. However, the correct and reliable operation of these processors is often compulsory, both in terms of consumer experience and for high-risk embedded domains such as medical and transportation systems. Thus, economical fault detection and recovery becomes essential to meet all necessary market requirements. This paper explores the efficient leveraging of superscalar, out-of-order architectures to enable multi-cycle transient fault-tolerance throughout the datapath in a novel manner. By using dynamic instruction execution redundancy, soft errors within the datapath are both detected and recovered. The proposed microarchitecture selectively reevaluates corrupted instructions, reducing the recovery impact by preserving completed instructions unaffected by the fault. The additional computational workload is dynamically staggered to leverage the out-of-order nature of the architecture and minimize resource conflicts and delays.

Categories and Subject Descriptors

B.8.1 [Performance and Reliability]: Reliability, Testing, and Fault-Tolerance

General Terms

Design, Reliability

Keywords

fault tolerance, soft errors, reliability, embedded, datapath

1. INTRODUCTION

As today's embedded processors continue Moore's perpetual trend of exponentially increasing transistor counts, the issue of transient hardware faults will become more prevalent. This is exacerbated by the smaller feature sizes, high

clock frequencies, and reduced voltage levels and noise margins being utilized to meet market demands. External events such as cosmic rays or ambient radiation can more easily alter voltage levels and the data represented by those levels [1]. This can lead to temporary inaccuracies within the data computations occurring within the processor, often termed single-event upsets (SEU), transient faults, or soft errors. In the past, the frequency of such transient faults was low, making fault-tolerant computers attractive only for high-risk, mission-critical domains such as medical devices, space programs, and ground and air transportation. However, future microprocessors will be more and more susceptible to transient faults as designs continue to grow in complexity and shrink in size [2, 3, 4]. As we begin to embark on 22nm technology designs, the impact of soft errors will become a significant factor in correct device operation and in the delivery of a positive consumer experience.

Furthermore, a glance into the future of embedded microprocessors will reveal an impending upheaval of present device technologies. Diminishing transistor-speed scaling and practical energy limits of current Silicon CMOS technology will soon compel a transition to a new technology scaling regime. Many arenas are currently under exploration, including carbon nanotubes, graphene, and quantum electronics. Such ground-breaking technologies will provide immensely higher device densities, but will also lead to substantially higher transient fault rates due to quantum effects, increased sensitivity to noise, and decreased fabrication tolerance [5].

As these next-generation processors are deployed into more ubiquitous arenas, such as automotive, industrial, and medical devices, the criticality of accurate computation becomes paramount. For example, a seemingly minute transient fault within a car's anti-lock braking system may result in a fatal accident. Indeed, the emergent domain of *dependable* microprocessors, such as the ARM Cortex-R series, can attest to the growing need for error-resistant processors even in the embedded domain.

Additionally, while fulfilling the goal of providing this fault tolerance, care must be taken in order to ensure that any associated overhead in terms of cost, performance, area, and power are not overly prohibitive. Simply duplicating or triplicating the entire system would indeed achieve resilience, but that would be quite inefficient. In order to remain competitive, a more intelligent and frugal approach to fault detection and recovery is necessitated. It has been shown that the soft error rate (SER) for SRAM cells decreases with decreasing technology size, and that the SER of latches remains relatively constant [6]. On the other hand, the SER of combinational logic within the processor increases quite rapidly as the feature size is reduced [6] and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CODES+ISSS'12, October 7–12, 2012, Tampere, Finland.
Copyright 2012 ACM 978-1-4503-1426-8/12/09 ...\$15.00.

as voltage is decreased [4]. Thus, given that memory arrays can affordably be hardened using parity, Hamming codes, or ECC since the cost of the coding logic is amortized over the array, focusing on hardening the combinational logic and state throughout the datapath will become the primary challenge.

This paper proposes a novel approach to leveraging a conventional superscalar, out-of-order architecture to efficiently enable fault-tolerance throughout the datapath, including multi-cycle SEUs. Redundancy is achieved by dynamically duplicating and independently executing each instruction within the datapath. The final result of each instruction pair is compared to detect soft errors occurring within the datapath. Upon fault detection, only the erroneous portion of the instruction chain (e.g. those instructions that were poisoned by the faulty computation) is selectively discarded. Other completed instructions that are independent and unaffected by the fault are preserved in order to reduce unnecessary re-computation time and energy. The duplicated instructions are dynamically staggered in order to leverage the out-of-order nature of the architecture and minimize resource conflicts and performance delays. This helps reduce structural hazards within the functional units, and allows for higher functional unit utilization and overall instruction throughput. We show the implementation of this architecture and provide experimental data taken over a general sample of complex, real-world benchmark applications to show the benefits of such an approach. The simulation results show full soft error recovery, while incurring only an 11% to 26% reduction in processor performance.

2. RELATED WORK

Prior research has investigated the inherent hardware redundancy provided by simultaneous multithreading (SMT) architectures. The AR-SMT architecture proposed duplicating an application into two instruction threads, one leading (A) and one trailing (R) [7]. Each instruction pair is compared and validated. Information from the leading thread is also used to assist the trailing thread with control and data predictions to reduce the performance penalty. Similarly, the authors in [8] leverage the SMT architecture, but relax the coupling between the two threads by only comparing those instructions that produce side-effects visible outside the processor core. In both these cases, fault detection is the primary contribution, with fault recovery involving much overhead.

From the perspective of chip multiprocessors (CMP), [9] proposed executing two duplicate application threads on different processor cores. In comparison to the SMT proposals, leveraging completely separate hardware in different cores obviates the possibility of a transient fault persisting long enough within a common hardware block to corrupt both threads. On the other hand, the inter-process communication to maintain and compare information across cores is costly. Similarly, the authors in [10] proposed executing two duplicate applications on different processor cores, but instead relied on the cache memory interface as the point of comparison and checkpointing to help alleviate the overheads of processor-to-processor communication.

Superscalar out-of-order (OOO) architectures have provided another appealing foundation for fault tolerance. The authors in [11] proposed duplicating instruction execution within the OOO processor, leveraging two adjacent re-order buffer (ROB) entries per instruction. Upon both copies completing execution, they are compared and, if identical, committed. On the other hand, if the pair differs then all the

ROB entries that have not been committed are discarded and execution is restarted from the last committed PC value. In this manner, the inherent rewind capability of the OOO processor allows fault recovery at no additional hardware cost. In a similar vein, the O3RS architecture proposed duplicating instruction execution, but removed the requirement of using two ROB entries per instruction [12]. Instead, they employ temporal redundancy by adding a differentiating state of *first* versus *second* execution to the ROB entries. In order to be committed, instructions must go from the *first* state to the *second* state and then be compared. If they differ, the state is moved back to *first* and subsequent ROB entries are also invalidated. Additionally, the idea of staggering the two computational threads was introduced in [13] in order to handle resource hazards, and showed improvement in the performance overhead.

Hybrid approaches of using an OOO processor coupled with an in-order redundancy pipeline have also been proposed. The DIVA architecture consists of a prototypical OOO processor, plus a very simple in-order *checker* processor [14]. Before instructions in the OOO pipeline can be retired, it must be compared with the result found from the in-order *checker*. While this improves the throughput (since the *checker* closely matches the throughput of the OOO processor), it requires separate functional units to be employed within the in-order *checker* which increases hardware overhead. The authors in [13] reduced this hardware overhead by allowing their in-order SHREC checker to share the same pool of functional units as the main OOO processor.

3. MOTIVATION

As mentioned in the introduction, combinational logic within the processor will increasingly become the victim of soft errors [6]. In particular, hardening the datapath of the processor is challenging due to the irregularity of the combinational logic used for computation. Whereas common fault-tolerance techniques, such as ECC, can be employed for uniform, array-like processor structures, the datapath poses additional complexity.

When discussing faults within the combinational logic of a circuit, it is important to observe that not every transient glitch will result in an SEU. The momentary voltage or current fluctuations that occur within the circuit are instead termed single-event transients (SETs). Only if the SET propagates through the circuit and results in an incorrect value being latched into a storage element does it become an SEU. Thus, the timing and longevity of the transient glitch can affect whether or not it manifests as an actual soft error or gets attenuated away. Because of this, the probability that momentary glitches will be captured as valid data in combinational logic increases linearly with frequency because the occurrence of clock edges increases [15, 16, 17].

Thus, the combination of ultra deep submicron scaling, high clock frequencies, and reduced voltage levels and noise margins all contribute to greatly increase the probability of transient glitches manifesting into SEUs within the processor datapath. Indeed, [18] reports the alpha-particle soft error rate (SER) measured increases of ≈ 2 -3x when reducing voltage from 0.95V to 0.75V in a 32nm circuit, while [4] reports a doubling in SER when reducing voltage from 0.7V to 0.5V in 28nm. Furthermore, the rate of propagation of a glitch through the circuit is dependent on the linear energy transfer (LET) of the particle strike. LETs as low as $3 \text{ MeV} \cdot \text{cm}^2/\text{mg}$ are capable of generating sizable transients, and an LET of $70 \text{ MeV} \cdot \text{cm}^2/\text{mg}$ can cause durations exceeding 1ns [17]. Since many high-end embedded

systems operate in multi-gigahertz frequencies, this implies that some SETs can last *more than one cycle* when induced by a particle with large enough LET. Given this, the consideration of guarding against multi-cycle faults within the datapath is also essential.

Fault tolerance requires both fault detection, as well as fault recovery once the fault is detected. There are numerous hardware techniques to detect whether a transient fault has occurred within a system. For example, one can simply create two duplicate systems and run them side-by-side, verifying the results at each time interval. Any differences would indicate the presence of a transient fault. While this approach would not degrade performance, the area and material cost of having two complete copies of the system is prohibitive. Furthermore, to enable fault recovery, a third system would typically be required. On the other hand, one can take a single system and execute the target application multiple times (whether in serial or in parallel), checking for differences. This temporal approach does not incur the area costs of hardware replication, yet the performance of such an approach would be poor. For embedded processors, the sensitivity to such area or performance overheads is often more pronounced and efficiency within the fault tolerant design is paramount.

In particular, there are three primary goals one should strive to attain to deliver robust and economic fault tolerance. First, given that the majority of the computations will not encounter a soft error, it is critical to minimize the performance impact of the fault detection logic for these non-faulty cases. Second, given the increased likelihood of soft errors occurring more frequently in ultra deep submicron feature sizes [2, 4], reducing the overhead of recovering from these faults is also important, both in terms of performance and scalability. Lastly, the assumption that transient faults only affect a single cycle is no longer true, necessitating the ability to handle multi-cycle faults. A novel architecture is proposed that will simultaneously achieve these three goals, intelligently leveraging a number of different design techniques to accomplish frugal multi-cycle transient fault tolerance within the datapath. An overview of this proposed architecture is discussed in the next section.

4. PROPOSED ARCHITECTURE

As previously mentioned, fault tolerant proposals often involve some form of spatial or temporal redundancy in order to detect and recover from faults. Such redundancy often incurs degradation in terms of area or performance. However, many of the inherent design properties of superscalar out-of-order (OOO) processors can be leveraged to help bridge this gap. In essence, these systems provide some duplication of hardware structures (functional units) and allow for multiple independent stream execution to occur in parallel. In order to detect transient faults, instruction redundancy can be leveraged within the OOO processor datapath. Instructions can be dynamically duplicated and independently executed. In this manner, fault detection within the datapath will simply become a matter of comparing the resulting values of the two instructions for differences. But, unlike full processor replication, many of the other hardware storage structures, such as the instruction queue, register file, caches, and TLBs, can remain shared and be hardened via other common techniques such as ECC or Hamming codes. Figure 1 shows the sphere of replication wherein the computation is duplicated. Only the computational datapath is replicated, greatly reducing the cost and area overhead of the fault detection. Furthermore, the same mechanisms in

Figure 1: Generic Tomasulo Architecture Showing Sphere of Replication

place for speculative out-of-order execution can be employed to enable fault recovery by simply causing a soft exception to occur and rewind the execution back to a valid prior state. Building on this speculative OOO architecture, the fulfillment of the three primary design goals is described in the following subsections.

4.1 Improving Non-Faulty Performance

One major concern about introducing fault tolerance is the associated performance penalty incurred regardless of whether faults occur. Simplistically in a processor that is 100% utilized, if every instruction is duplicated, then the performance would degrade to 50% of the non-duplicating baseline. However, all computational resources of a processor are rarely utilized 100% continuously. In reality, many functional units may be idle at any given time. In order to reduce the impact of instruction duplication, one would ideally wish to have the duplicated instructions execute on idle functional units, and thus avoid elongating the time of the primary instruction thread.

Yet, if one simply creates two duplicate, independent execution threads, the functional unit access patterns will be identical. This will lower the likelihood of being able to perfectly interleave the secondary thread's hardware needs with that of the first's. The differing lengths of time each functional unit class takes (e.g. floating-point divide vs. integer add) will prevent a solution of simply offsetting the secondary thread by a fixed period of time. Furthermore, once the secondary thread begins, it may then occupy a resource that is also needed by the primary thread, which will elongate the overall time. An example of this is shown in Figure 2(a). The primary thread (shown in blue) and the secondary thread (shown in red) are independent. Assume there is only one functional unit of type *A* and it takes two cycles to complete, while there are two functional units of type *B* and they take one cycle to complete. As one can see,

Figure 2: Duplicate Instruction Thread Decoupling Improves Performance

this would force both threads to be sequential, causing the overall time to take 6 cycles.

To rectify this inefficiency, a decoupling of the secondary thread is proposed. Duplicate secondary instructions are still created, but the source operands for each secondary instruction are connected to the corresponding primary instruction(s). Figure 2(b) shows this proposed modification. As one can see, because of this extra degree of freedom, the processor can complete these same six instructions in four cycles instead of six. The hardware utilization is improved, since the additional B -type functional units are now leveraged instead of waiting idly. In this fashion, one can ameliorate and amortize the performance overhead of instruction duplication (regardless of the presence of faults) by decoupling the secondary instructions and allowing them to execute at a lower priority in any functional units that may be available.

4.2 Improving Faulty Performance

As fault rates increase, the importance of optimizing the fault recovery aspect of the proposed architecture becomes more important. Assuming a transient fault has been detected, the most basic recovery approach for a speculative out-of-order processor would be to rewind the state to just before the offending instruction executes, and then restart execution from that point. This effectively throws away all computation that occurred after the offending instruction. This behavior can be quite wasteful if the number of subsequent instructions computed after the instruction in question is large, and those computed instructions are *unaffected* by the offending instruction.

Instead, upon fault detection, only the erroneous portion of the instruction chain (e.g. those instructions that were poisoned by the faulty computation) should be selectively discarded. Other completed instructions that are independent and unaffected by the fault are preserved in order to reduce unnecessary re-computation overhead. This not only abates resource contention within the system, but also enjoys the added benefit of reducing power consumption. Furthermore, as the occurrence of transient faults increases, the proposed architecture will be able to more robustly handle recovery without becoming overwhelmed and unable to re-

Figure 3: Impact of Uniformly Distributed Fault Results in Balanced Average Re-Computation Effort

solve faults faster than they occur. Thus, this approach will be able to scale with increases in soft error frequency.

In addition to employing poison bits to localize the re-computation effort and avoid invalidating unaffected work, a further optimization is proposed to try and recuperate potentially poisoned work. Given that there are two differing copies of the offending instruction, but no knowledge of which one is valid, a third computation of the offending instruction is initiated and the result of that computation acts as a vote in favor of one of the other two copies. In this manner, the third computation can select which of the two original copies is the correct one, and preserve the valid computational work that was performed in subsequent instructions. The erroneous copy of the instruction is discarded, as well as any subsequent instructions that had the corresponding poison bits.

Assuming a uniform fault distribution, a fault could occur within a primary or secondary instruction with equal probability. Yet, if the fault occurred in the primary instruction, the poison bit setup would impact both the dependent primary instructions, as well as the dependent secondary instructions. On the other hand, if the fault occurred in the secondary instruction, only that single instruction would be affected (since all secondary instructions pull their source operands from primary instructions). As one can observe, probabilistically these two cases balance out. Half of the time, the fault will be in a primary instruction, which would impact the entire dependency chain and require those instructions to be re-computed, as shown in Figure 3(a). The other half of the time, it will occur in the secondary instruction and that single instruction can just be discarded at that point, as shown in Figure 3(b).

4.3 Handling Multi-Cycle Faults

The main concern with multi-cycle faults is the condition where both the primary and secondary instructions are executed on the same functional unit while that unit is being perturbed by the multi-cycle fault. In this situation, both copies of the instruction may match, but both may be incorrect. Thus, the fault detection capability of the system becomes undermined. In order to rectify this situation, each functional unit must guard against being selected to compute an instruction that had the complement instruction previously executed on it. This will ensure the same functional unit will not be used by both copies of the same instruction. One challenge is if the system only has a single functional unit, as may be the case for more infrequent and expensive calculations such as floating-point division (FDIV). In these cases, since there is only a single

functional unit, it must be utilized by both the primary and secondary instructions. As will be shown in Section 6, a temporal factor will be included in the guard logic to allow a single functional unit to be shared by both the primary and secondary instructions.

5. DESIGN DECISIONS

While early OOO implementations had a central register update unit (RUU) for bookkeeping, most modern implementations decentralize the information and have a separate reservation station, ROB, and register file to improve throughput in the presence of speculation. This latter class of speculative out-of-order architectures is derived from the Tomasulo design [19], and is the foundation of our proposed fault tolerant implementation. Furthermore, this proposal focuses on the datapath of the processor, which is more challenging to protect due to its irregularity when compared to array-like storage structures. The other processor components, such as cache memories, register files, and TLBs, are assumed to be hardened against faults using existing solutions, such as ECC, and will not be discussed further.

While the original Tomasulo architecture only scheduled instructions via the issue queue read from instruction memory, enabling faulty instructions to re-execute will require a mechanism to reintroduce a completed instruction back into the execution engine. The detailed modifications that are done to implement this are discussed in the next section.

Instructions that cause external side-effects, such as loads, stores, and conditional branch instructions will not be fully duplicated. Allowing two copies of such instructions to execute can cause serious performance and correctness issues within the pipeline. For example, having a load instruction duplicated may cause undesirable cache implications, such as evicting other entries that were also active. Additionally, the bandwidth of the memory subsystem is often a limiting factor and unnecessary congestion should be avoided. Thus, instead of fully duplicating memory instructions, only the effective address calculation is hardened via this proposed duplication scheme, but the actual memory access operation occurs only once. This avoids memory consistency issues and possible performance degradation due to cache sensitivities. As modern architectures typically harden the memory via ECC or similar safeguards, it can be assumed that correct values are retrieved from or stored into memory cells with no additional effort. Similarly, having two copies of a conditional branch instruction that may be divergent will cause many issues. Given this, branch instructions are also not fully duplicated. Rather, only the conditional computation within the datapath is duplicated, but the actual modification of the program-counter is only done once. Again, it is presumed that storage elements providing the target address are hardened against faults using typical memory-array fault tolerance techniques.

6. IMPLEMENTATION

The typical reservation station entry contains the following fields:

- *OpCode* - operation to be performed
- V_j - value of operand 1
- V_k - value of operand 2
- Q_j - ROB # of operand 1 (0 if already available)
- Q_k - ROB # of operand 2 (0 if already available)
- *Dest* - destination ROB #
- *A* - effective address for load/store
- *Busy* - indicates entry and corresponding FU occupied

Reservation Station Entry

OpCode	V_j	V_k	Q_j	Q_k	Dest	A	Busy	Type
6 bits	32 bits	32 bits	8 bits	8 bits	8 bits	32 bits	1 bit	2 bits

Figure 4: Modifications to Reservation Station Entries

Reorder Buffer Entry (Original)

InstType	Dest	Value	Ready
2 bits	32 bits	32 bits	1 bit

OpCode	Dest	Poison	Q_j	Q_k	Value0	Value1	Ready0	Ready1
6 bits	32 bits	256 bits	8 bits	8 bits	32 bits	32 bits	1 bit	1 bit

Reorder Buffer Entry (Modified)

Figure 5: Modifications to Reservation Station Entries

For this proposal, an additional 2-bit field is needed per reservation station entry:

- *Type* - indicates if this is *primary*(0), *secondary*(1), or *tertiary*(2) instruction

The necessary modifications to the reservation station entries are shown in Figure 4, highlighted in red. Moreover, the typical ROB entry contains four fields:

- *InstructionType* - branch, store, or ALU/load
- *Dest* - register or memory address
- *Value* - value of instruction result
- *Ready* - indicates if *Value* is ready

However, in order to enable this proposal each ROB entry will instead contain:

- *OpCode* - operation to be performed
- *Dest* - register or memory address
- *Poison* - bit-vector of progenitor ROB entries
- Q_j - ROB or RF # of operand 1
- Q_k - ROB or RF # of operand 2
- *Value*[2] - value of instruction result
- *Ready*[2] - indicates if *Value* is ready

The necessary modifications for each ROB entry are shown in Figure 5, again highlighted in red. The bit numbers shown assume an ROB size of 256 entries, and can be adjusted accordingly.

The instruction stream will dynamically be duplicated within the computational datapath, running independently on the available hardware. Figure 1 shows the sphere of replication wherein the computation is duplicated. Once an instruction is issued, two independent computations will exist for that instruction. Subsequent dependent instructions that are also duplicated will receive both of their operands from only the primary thread (or the register file, if already committed). In this manner, the additional computation overhead of the second instruction can be decoupled and executed whenever there are idle computation resources available. The pair of instructions will occupy a single ROB

Figure 6: Example of Selective Poisoning in Both Primary and Secondary Threads

entry, but will account for two reservation station entries. Within a given ROB entry, the new *Poison* field that is added will indicate all the other ROB entries that, if erroneous, may potentially pollute this entry's source operands and cause an error. The *Value* and *Ready* fields are duplicated to hold the results of both the *primary* and *secondary* instruction results received from a functional unit. When a functional unit completes its computation, it will send the result value along with the destination ROB # and whether it was a *primary* or *secondary* instruction, allowing the correct ROB entry *Value* field to be updated.

Once both copies of the instruction are completed (e.g. both *Ready* fields in the ROB entry are true), they are compared to detect if a transient fault has occurred. If they match, the instruction is ready to be committed by writing the computed value (either from the *primary* or *secondary Value* field within the ROB) into the destination register. This commit will only occur once the ROB entry reaches the *head* of the buffer (to ensure in-order commit, as normally expected). Once the commit occurs, any other ROB entries that refer to the committed entry in their Q_j or Q_k fields will update their index to instead point to the register file destination that was committed by the ROB. In this manner, if a subsequent ROB entry needs to be re-executed (due to a mismatch within itself or from one of its progenitor instructions), the Q_j and Q_k values will be used to correctly repopulate a reservation station entry to allow for the re-execution of that particular ROB entry.

On the other hand, if the two values mismatch, a fault has been detected. In this situation, a third, independent copy of the mismatched instruction will be submitted as a new entry into the reservation station, utilizing the information stored locally within the ROB fields. The reservation station will treat this tertiary instruction as the highest priority, assigning it into the first available functional unit that is able to service the computation. Once the result is computed, it will be compared against both existing *Value* fields in the ROB for a match. If neither matches, another third copy will be submitted and the process recurs until the resulting value matches one of the two existing *Value* fields. An optional maximum retry limit can be added to account for the case of an unrecoverable fault.

Once a match against the third thread is found, that thread (be it *primary* or *secondary*) will be considered correct and the other thread invalid. If it is the *primary* thread that is deemed invalid, then the offending ROB entry will broadcast its ROB # to the other ROB entries. The others

will perform a simple bitwise *AND* against their *Poison* field to detect if they may have been poisoned by the fault. The offending ROB entry and all those ROB entries that were poisoned will clear their *Ready* fields and resubmit a new pair of entries into the reservation station for both *primary* and *secondary* instructions, utilizing the information stored locally within the ROB fields. If it is the *secondary* thread that is deemed invalid, no extra work is incurred and the ROB entry is now ready to be committed. The superiority of this approach can be appreciated in that statistically half of the times that a mismatch occurs, the ROB entries are preserved and do not need to be re-computed, reducing the performance and power overhead of fault recovery (since the erroneous entry was the *secondary* instruction). To illustrate this concept, Figure 6 shows (a) an example data-flow graph (DFG) of six instructions, (b) the case of a fault occurring in a *primary* instruction resulting in the poisoning of data-dependent instructions, and (c) the case of a fault occurring in a *secondary* instruction which does not effect data-dependent instructions.

The reservation station functional unit allocation algorithm is modified to allow it to service instructions in a fixed-priority fashion. *Tertiary* instructions (those that have been spawned in order to vote on the correctness of either the *primary* or *secondary* instruction) are considered the highest priority. *Primary* instructions are considered the next highest priority, and *secondary* instructions are considered the lowest priority. The reservation station will try to assign the highest priority instruction it can find that has both of its operand values ready into the next available functional unit. If multiple equivalent-priority instructions are ready to be executed, one is selected at random. An important observation is that this priority-based allocation scheme avoids deadlock. While *secondary* instructions may be likely to get stuck waiting for a functional unit if there are many *primary* instructions continuously occupying the functional units, the general structure of the ROB imposes a dampener on this resource starvation. No new instructions can be issued into the ROB if all the ROB entries are full. If *secondary* instructions are being held in the reservation stations, they preclude the retirement of their corresponding ROB entries. Thus, at some point the ROB will be full and no new *primary* instructions will be issued. This will cause the stream of *primary* instructions to wane and allow the awaiting *secondary* instructions to execute. Once the *secondary* instructions finish and the ROB entries start to commit, then new ROB entries will become available and the system can continue to move forward.

The existence of multi-cycle faults necessitates some additional intelligence in the reservation stations. Utilizing the same exact functional unit for the complement pair of instructions can allow multi-cycle faults to undermine the fault tolerance of the system. To rectify this, the reservation station must ensure both the *primary* and *secondary* instruction do not execute on the same functional unit, or at least not within a certain time period. This is accomplished by having a small table within the reservation station that keeps track of the last N instructions that were executed on each functional unit. The value of N can be determined by the likelihood of a multi-cycle fault lasting longer than N functional unit calculations. For example, if it is infeasible for a fault to last longer than 2 functional unit calculations (e.g. SUB or FDIV), then the table will contain just two entries. The entries will just need to hold the ROB # plus the 2-bits of *Type* to determine which type of instruction it was (*primary*, *secondary*, or *tertiary*). Additionally, to sup-

Table 1: Hardware Config 1

ROB Entries	256
Reservation Station Entries	32 per FU type
LSQ Entries	64
Instruction L1 cache	64 KB, 2-way set-associative
Data L1 cache	32 KB, 2-way set-associative
Unified L2 cache	512KB, 4-way set-associative
Functional Unit Mix	4 IALU (1), 2 IMUL/IDIV (3/19), 2 FALU (2), 2 FMUL/FDIV (4/12)

port the case of a single functional unit (e.g. for FDIV), a time-based decay is added to remove the table entries after a period of inactivity greater than the anticipated multi-cycle fault length. In this manner, full fault tolerance against multi-cycle faults is accomplished, while incurring minimal overhead in the regular operation of the entire datapath.

Based on the proposed architectural modifications, each reservation station would need 2 bits per entry in addition to the preexisting 127 bits. Similarly, each ROB entry would require an additional 309 bits on top of the preexisting 67 bits. Given an architecture with 256 ROB entries and 128 total reservation station entries (for all types of functional units), this would amount to approximately 9.7KB of additional storage elements. While this amount of data is not trivial, it is quite a reasonable price to pay to enable transient fault detection and recovery throughout the processor datapath and demonstrates the frugality of the proposed fault tolerant design.

7. EXPERIMENTAL RESULTS

In order to assess the benefit from this proposed architectural design, we utilized the SimpleScalar framework [20]. The stock code initially utilized a basic register update unit (RUU) structure, combining the reorder buffer (ROB) and reservation stations and provided no register renaming. In order to fully exploit possible parallelism and instruction throughput, we greatly modified the default *sim-outorder* simulator to implement a full speculative Tomasulo architecture [19], including register renaming and decentralized reservation stations. This will allow hardware instruction scheduling to cross basic block boundaries and implicitly reduces register pressure, both of which will help improve IPC (instructions per cycle). Furthermore, the simulator is augmented with the fault detection and recovery scheme presented in this paper. When operating in fault-tolerant mode, all datapath instructions will be replicated and validated. For instructions with side-effects, such as memory accesses and control flow operations, only the numerical computation (e.g. effective memory address calculation or branch condition comparison) will be replicated; the actual modification of the LSQ (load/store queue) or program counter will only be permitted from the primary instruction. This behavior ensures program validity; if a fault does occur the precise exception handling of the speculative Tomasulo implementation will be able to correctly rewind the system state.

A random fault injection routine was added to the simulator, allowing a configurable rate at which to randomly corrupt instructions during execution. As each instruction begins execution within a functional unit, the fault injection routine is queried to determine if that instruction should be forced to be corrupted. If so, the resulting value from the function unit is *XOR*ed with -1 , effectively flipping all the bits. Furthermore, 20% of the instances where the fault injection routine triggers a fault, it will also mark the corresponding functional unit as faulty for 2 instructions, simulat-

Table 2: Hardware Config 2

ROB Entries	128
Reservation Station Entries	16 per FU type
LSQ Entries	64
Instruction L1 cache	64 KB, 2-way set-associative
Data L1 cache	32 KB, 2-way set-associative
Unified L2 cache	512KB, 4-way set-associative
Functional Unit Mix	4 IALU (1), 2 IMUL/IDIV (3/19), 2 FALU (2), 1 FMUL/FDIV (4/12)

ing a multi-cycle fault. In these cases, the next instruction that enters the functional unit will also be corrupted in the exact same fashion.

We chose two representative superscalar out-of-order system configurations to demonstrate our proposal. Table 1 defines *Config 1*, which is a more aggressive system employing a 256-entry ROB and 32-entry reservation stations for each functional unit type. Table 2 defines *Config 2*, which is more conservative and has half the number of ROB and reservation station entries, and also only a single floating-point multiplier and divider (FMUL/FDIV). Both configurations have pipelined functional units (except for the IDIV and FDIV units). The number of cycles each functional unit type consumes is listed within the parentheses in each table.

The complete SPEC CPU2000 benchmark suite [21] is used, providing 12 integer and 14 floating-point real-world applications. A listing of these benchmarks and their respective descriptions are provided in Table 3. The benchmarks are cross-compiled for the PISA instruction set using the highest level of optimization available for the language-specific compiler. The reference inputs are used for each benchmark, with each benchmark executed in its entirety from start to finish.

In order to assess our proposed architecture, we simulated the benchmarks on four different processor models: *Baseline*, *FT No Faults*, *FT 1E-4 Fault Rate*, and *FT 1E-1 Fault Rate*. The *Baseline* processor model is a plain speculative Tomasulo architecture [19], with no instruction replication or fault tolerance enabled. The *FT No Faults* processor model enables instruction replication and fault tolerance

Table 3: Description of Benchmarks

Benchmark	Description
bzip2	Memory-based compression algorithm
crafty	High-performance chess playing game
I eon	Probabilistic ray tracer visualization
N gap	Group theory interpreter
T gcc	GNU C compiler
E gzip	LZ77 compression algorithm
G mcf	Vehicle scheduling combinatorial optimization
E parser	Dictionary-based word processing
R perlbnk	Perl programming language interpreter
twolf	CAD place-and-route simulation
vortex	Object-oriented database transactions
vpr	FPGA circuit placement and routing
F ammp	Computational chemistry
L applu	Parabolic partial differential equations
O apsi	Meteorology pollutant distribution
A art	Image recognition / neural networks
T equake	Seismic wave propagation simulation
I facerec	Face recognition image processing
N fma3d	Finite-element crash simulation
G galgel	Computational fluid dynamics
lucas	Number theory / primality testing
P mesa	3D graphics library
O mgrid	Multi-grid 3D potential field solver
I sixtrack	High energy nuclear physics accelerator
N swim	Shallow water modeling
T wupwise	Physics / quantum chromodynamics

Figure 7: Performance Impact of Fault Tolerance on Integer Benchmarks using *Config 1*

checking, but will not inject any faults. This will demonstrate the overhead of our proposed fault tolerance scheme without the additional cost of recovering from faults. The *FT 1E-4 Fault Rate* and *FT 1E-1 Fault Rate* processor models also enable the fault tolerance scheme, but will inject faults at the rate of 0.01% and 10%, respectively. The experiment range is selected broadly so as to demonstrate the effectiveness of our design in enduring a variety of fault rates, including high ones.

Figure 7 compares the IPC of the four processor models using *Config 1* for each of the 12 integer benchmarks. As one can see, the Baseline attained an average IPC of approximately 1.77. When we enable the fault tolerance datapath replication, the average IPC falls to 1.57, which is a reduction of about 11%. Given that every computational instruction is being executed twice, such a low reduction in IPC is quite impressive. Indeed, if every functional unit was continuously utilized throughout the lifetime of the program, such instruction duplication would amount to an IPC reduction of at least 50%. However, due to the intelligent and efficient implementation of our system, the duplicated instructions are able to dynamically recuperate idle hardware resources and achieve improved overall IPC while doing double the number of computations.

Furthermore, the introduction of a 0.01% fault rate does not cause any noticeable decline in IPC. In fact, the overhead of recovering these faults is quite minimal due to the selective fault chaining that was implemented. When the *primary* and *secondary* instructions mismatch and the *tertiary* voter instruction determines which of the two is incorrect, then only the erroneous instruction and those instructions that were poisoned by it will be thrown away and re-computed. This allows for many independent instructions that were executed in between to be preserved, as opposed to just rewinding the entire ROB back to the offending instruction and starting over. Furthermore, even the re-computation of the poisoned chain is something that will be avoided 50% of the time; if the fault is determined to have occurred in the *secondary* instruction, no re-computation is necessary. This not only helps reduce the performance overhead of fault recovery, but also helps curb unnecessary power consumption.

This efficiency becomes even more apparent as the fault

Figure 8: Performance Impact of Fault Tolerance on Floating-Point Benchmarks using *Config 1*

rate goes up to 10%. In the *FT 1E-1 Fault Rate* model, we can still see the system is able to make forward progress, even though about 1 out of every 10 instruction executions was faulty, plus 2 of those times the fault was multi-cycle and corrupted the next instruction. While it may seem like an extreme case to have such a high fault rate, it is important to demonstrate the ability of the system to adeptly handle high rates of failure.

Figure 8 compares the IPC for the same four processor models using *Config 1* across the floating-point benchmarks. In general, the floating-point benchmarks are able to achieve a higher IPC, as their diversity of additional floating-point instructions helps reduce the demands on the other functional units. The average IPC for the Baseline in this case was 2.54, while the *FT No Faults* model achieved 1.87. This reflects a reduction in IPC of approximately 26%. As one can see, the IPC reduction is more pronounced in the floating-point benchmarks. This is due to the smaller number and longer execution time of floating-point functional units, which can become a bottleneck as two copies of each instruction need to be computed. The overhead is quite low, considering the doubling in the computational work that is taking place.

Similar to the integer benchmarks, introducing a modest amount of faults does not significantly reduce the average IPC for floating-point applications. One is able to achieve complete datapath fault tolerance, while only incurring a 26% reduction in performance. Furthermore, the system is still able to handle the high level of faults of the *FT 1E-1 Fault Rate* processor, demonstrating that even in the face of extreme fault occurrence the system is able to make forward progress and eventually provide correct values.

Figure 9 and Figure 10 show the same IPC information for both integer and floating-point applications, respectively, but using the more conservative *Config 2*. In these cases, the number of ROB entries and reservation stations is reduced by 50%. This will reduce the amount of inherent IPC by constraining the amount of instructions that can be in-flight, and also shrinking the effective window of how far the *primary* instruction thread can run ahead of the *secondary* instructions before having to wait for the *secondary* instructions to catch up. These resource limitations will also

Figure 9: Performance Impact of Fault Tolerance on Integer Benchmarks using *Config 2*

further degrade the performance when instruction replication is enabled, since the available slack in the system is more constrained and the availability of empty reservation station entries is diminished.

As one can see, the *Baseline* IPC for integer and floating-point benchmarks using *Config 2* was 1.53 and 1.72, respectively. When instruction replication is enabled without any faults, the IPC values are reduced by about 18% and 37% to become 1.26 and 1.09, respectively. Once fault injection is enabled, the IPC does slightly degrade as would be expected due to the instruction re-computation overhead. Furthermore, there may be an additional delay in sending the *tertiary* instruction to the reservation station for execution if all entries in the smaller reservation station are already occupied.

Additionally, *Config 2* demonstrates the effectiveness of our proposal in situations where only a single functional unit may exist. In this configuration, only a single FMUL and FDIV functional unit is available. In the case where a multi-cycle fault occurs during an FDIV computation of a *primary* instruction, the system must be smart enough to guard against the corresponding *secondary* instruction running immediately afterward. If that situation were allowed to happen, the *secondary* instruction would be corrupted in an identical fashion, and the fault detection would fail to cover that case. In executing the various benchmarks on *Config 2*, we were able to verify that the architecture properly guarded against this situation, and only allowed a corresponding instruction to execute on the same functional unit after the necessary delay period to ensure the multi-cycle fault would be over.

In addition to the aforementioned IPC values computed for the two fault rates of 1E-4 and 1E-1, we were interested to see the impact on IPC as a result of increasing fault rates. We chose 15 different fault rates, ranging from 1E-6 to 1E-1 to examine how rapidly IPC degrades as the fault rate increases. Obviously, at some point the fault rate will outpace the ability of the system to recover, causing faults faster than they can be re-computed. As this point is approached, one would expect to see an exponential drop-off in IPC. A group of six benchmarks were chosen to explore this behavior: *perlbnk*, *gap*, and *crafty* from the integer suite, and *sixtrack*, *apsi*, and *applu* from the floating-point suite.

Figure 10: Performance Impact of Fault Tolerance on Floating-Point Benchmarks using *Config 2*

Using the *Config 1* setup, these benchmarks were executed while varying the fault rate.

Figure 11 presents the interaction of the increasing fault rate on the IPC for the three integer benchmarks. As one can see, the IPC remains somewhat stable with fault rates below 1E-3 (about 1 instruction every 1,000 instructions executed). As the fault rate increases beyond that level, the degradation of IPC becomes more pronounced and eventually begins to rapidly plunge as the fault rate becomes increasingly frequent. When the fault rate reaches 1E-1 (about 1 instruction every 10 instructions executed), the IPC is on a sharp trajectory to soon become stalled and unable to move forward due to overwhelming faults.

Figure 12 presents the same interaction of IPC versus fault rate, but using the three floating-point benchmarks. Similar to the integer data, the IPC remains about the same until reaching 1E-3. At that point, the IPC decays exponentially and eventually will result in the same livelock situation where faults occur faster than they can be recovered from.

Figure 11: Integer IPC versus Fault Rate

Figure 12: Floating-Point IPC versus Fault Rate

8. CONCLUSIONS

As the feature size and operating voltages of high-end embedded processors continue to shrink to satisfy consumer's insatiable demand for greater functionality and less power consumption, sensitivity to soft errors increases dramatically. In the upcoming generation of 22nm devices and below, the transient fault rate in high-performance processors will increase noticeably. Given this, the correct and reliable operation of these microprocessors will become a more prominent objective in order to satisfy consumer experience levels. Furthermore, as these next-generation processors are deployed into more ubiquitous arenas, such as automotive, industrial, and medical devices, the criticality of accurate computation becomes paramount. Moreover, it has been shown that the combinational logic of the datapath will become a major contributor of soft errors, and an efficient solution to provide fault tolerance is necessary.

In this paper, we have presented a novel and frugal fault tolerance framework for current superscalar out-of-order architectures. In particular, the system is able to account for multi-cycle SEUs, which many existing approaches do not guard against. Furthermore, by leveraging most of the existing hardware structures of the out-of-order machine, the entire datapath can be hardened against transient faults with minimal additional hardware costs. Redundancy is achieved by dynamically duplicating and independently executing each instruction within the datapath's out-of-order infrastructure. Each pair of instructions is compared to detect soft errors, and recovery is accomplished by selectively re-executing only the necessary instructions that were poisoned by the fault. A key optimization is that other completed instructions that were independent and unaffected by the fault are preserved in order to reduce unnecessary re-computation overhead. Furthermore, in order to maximize the IPC during non-faulty operation, the duplicated instructions are allowed to be dynamically staggered and utilize idle hardware functional units in an out-of-order fashion to reduce resource conflicts and improve IPC.

Extensive experimental results are provided to validate the correct operation of the architecture. Complete fault detection and recovery is achieved within the processor datapath, while only incurring moderate reductions of IPC of 11% to 26% in order to handle the replicated computations. Furthermore, the system is able to withstand very high rates

of soft errors, and only begins to significantly degrade performance as the fault rate reaches 10% of instructions executed.

9. REFERENCES

- [1] R. Baumann. Soft errors in advanced computer systems. *IEEE Design and Test of Computers*, 22(3):258–266, 2005.
- [2] K. Lee, A. Shrivastava, M. Kim, N. Dutt, and N. Venkatasubramanian. Mitigating the impact of hardware defects on multimedia applications: A cross-layer approach. In *Proc. Intl. Conf. on Multimedia*, pages 319–328, 2008.
- [3] N. Seifert, V. Ambrose, B. Gill, Q. Shi, R. Allmon, C. Recchia, S. Mukherjee, N. Nassif, J. Krause, J. Pickholtz, and A. Balasubramanian. On the radiation-induced soft error performance of hardened sequential elements in advanced bulk cmos technologies. In *Proc. Intl. Reliability Physics Symp.*, pages 188–197, 2010.
- [4] A. Dixit and A. Wood. The impact of new technology on soft error rates. In *Proc. Intl. Reliability Physics Symp.*, pages 486–492, 2011.
- [5] J. Han, J. Gao, P. Jonker, Q. Yan, and J. A. B. Fortes. Toward hardware-redundant, fault-tolerant logic for nanoelectronics. *IEEE Design and Test of Computer*, 22(4):328–339, 2005.
- [6] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. Intl. Conf. on Dependable Systems and Networks*, pages 389–398, 2002.
- [7] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proc. 29th Intl. Symp. on Fault-Tolerant Computing*, pages 84–91, 1999.
- [8] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proc. 27th Intl. Symp. on Computer Architecture*, pages 25–36, 2000.
- [9] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proc. 30th Intl. Symp. on Computer Architecture*, pages 98–109, 2003.
- [10] C. Yang and A. Orailoglu. A light-weight cache-based fault detection and checkpointing scheme for MPSoCs enabling relaxed execution synchronization. In *Proc. Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, pages 11–20, 2008.
- [11] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proc. 34th Intl. Symp. on Microarchitecture*, pages 214–224, 2001.
- [12] A. Mendelson and N. Suri. Designing high-performance & reliable superscalar architectures: The out of order reliable superscalar (O3RS) approach. In *Proc. Intl. Conf. on Dependable Systems and Networks*, pages 473–481, 2000.
- [13] J. C. Smolens, J. Kim, J. C. Hoe, and B. Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proc. 37th Intl. Symp. on Microarchitecture*, pages 257–268, 2004.
- [14] T. M. Austin. Diva: A reliable substrate for deep submicron microarchitecture design. In *Proc. 32nd Intl. Symp. on Microarchitecture*, pages 196–207, 1999.
- [15] S. Buchner, M. Baze, D. Brown, D. McMorrow, and J. Melinger. Comparison of error rates in combinational and sequential logic. *IEEE Transactions on Nuclear Science*, 44(6):2209–2216, 1997.
- [16] L. Anghel and M. Nicolaidis. Cost reduction and evaluation of temporary faults detecting technique. In *Proc. Intl. Conf. on Design, Automation and Test in Europe*, pages 591–598, 2000.
- [17] P. E. Dodd, M. R. Shaneyfelt, J. A. Felix, and J. R. Schwank. Production and propagation of single-event transients in high-speed digital logic ICs. *IEEE Transactions on Nuclear Science*, 51(6):3278–3284, 2004.
- [18] B. Gill, N. Seifert, and V. Zia. Comparison of alpha-particle and neutron-induced combinational and sequential logic error rates at the 32nm technology node. In *Proc. 47th Intl. Symp. on Reliability Physics*, pages 199–205, 2009.
- [19] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Fifth Edition, 2011.
- [20] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, pages 59–67, 2002.
- [21] SPEC CPU2000 Benchmarks. <http://www.spec.org/cpu/>.