Ditto Processor

Shih-Chang Lai
Dept. of ECE
Oregon State University
Corvallis, OR

Shih-Lien Lu
Konrad Lai
Microprocessor Research,
Intel Labs.
Hillsboro, OR
Shih-lien.l.lu@intel.com

Jih-Kwon Peir
Dept. of Computer Science
University of Florida
Gainesville, FL

Abstract

Concentration of design effort for current single-chip Commercial-Off-The-Shelf (COTS) microprocessors has been directed towards performance. Reliability has not been the primary focus. As supply voltage scales to accommodate technology scaling and to lower power consumption, transient errors are more likely to be introduced. The basic idea behind any error tolerance scheme involves some type of redundancy. Redundancy techniques can be categorized in three general categories: (1) hardware redundancy, (2) information redundancy, and (3) time redundancy. Existing time redundant techniques for improving reliability of a superscalar processor utilize the otherwise unused hardware resources as much as possible to hide the overhead of program re-execution and verification. However, our study reveals that re-executing of long latency operations contributes to performance loss. We suggest a method to handle short and long latency instructions in slightly different ways to reduce the performance degradation. Our goal is to minimize the hardware overhead and performance degradation while maximizing the fault detection coverage. Experimental studies through microarchitecture simulation are used to compare performance lost due to the proposed scheme with non-fault tolerant design and different existing time redundant fault tolerant schemes. Fourteen integer and floating-point benchmarks are simulated with 1.8~13.3% performance loss when compared with non-fault-tolerant superscalar processor.

1. Introduction

Transient errors, also called soft errors, can be introduced by alpha or neutron particles strikes. They can also be introduced by power supply disturbances or other environmental variations. As supply voltage scales to accommodate technology scaling and to lower power consumption, transient errors are more likely to be introduced [1-5]. Transient errors may affect microprocessors in many ways [6-7]. One possible manifestation of soft errors in the modern processor is undetected data corruption. Experiments done by injecting faults into unprotected microprocessors resulted in the observation of non-negligible risk of data corruption [8]. Soft errors cannot be detected by manufacturing testing nor by periodic testing. With widespread usage of microprocessors in critical financial data processing, it is desirable to have microprocessors capable of transparent recovery and protection from data corruption in the face of soft errors.

The basic idea behind any error tolerance schemes involves some type of redundancy. Redundancy techniques can be categorized in three general categories [9-10]: (1) hardware redundancy [11], (2) information redundancy [13-14] and (3) time redundancy [15-16]. Hardware redundancy employs physical duplication and achieves redundancy spatially. Information redundancy with error detection and correction coding is effective in protecting memory elements against transient faults. Transient faults that occur in the logic blocks have no easy way to increase immunity besides utilizing either hardware redundancy or time redundancy. Time redundancy re-executes operations with the same hardware and obtain redundancy temporally. Time redundancy can be performed at different levels of the microprocessor. Work done by Nicolaidis [17] proposed a way to duplicate in time at the circuit level. This method introduces a delay element between the combination logic and the pipeline register allowing the data to be latched twice at different time. At the microarchitecture level, time redundancy can be achieved by instruction re-execution or by check pointing and rollback [18]. At the software level it can be accomplished by statically duplicating the program in multiple versions [12]. It assumes that if one version fails, other versions will produce correct results. In this paper, we focus on the microarchitecture level of time redundancy technique.

Existing microarchitecture level time redundancy mechanisms lose performance due to blindly duplicating the execution of instructions at either decode stage [25][27][32][34] or at commit stage [22-24][33]. Both schemes verify the result at the point when the original copy is ready to retire and redundant copy has completed execution.

1. Duplicating the instructions at decode stage generates many unnecessary instructions to consume hardware resource when branch mis-prediction occurs.
2. The second scheme stored the committed instructions to a buffer in program order. This buffer provides the information of retired instructions to the fetch units. The instructions would then be re-fetched, re-decoded, re-renamed and re-executed.

The main drawback of the first scheme is that it does not cover faults that may occur at the frond-end of the pipeline. The second approach is commonly used in Simultaneous multithreading (SMT) based fault tolerant processors. They have better fault coverage compare to the first approach. However, if they only have limited resources, the performance degradation of the second scheme is worse than the first. The main reason is that the second scheme reduced the instruction bandwidth available to the original instruction stream [27]. Since long latency operations tend to stay in reorder buffer longer than short latency operation, our study reveals that the long latency operations are important factor to the performance loss of both schemes. Here we categorize memory reference micro-operation, multiply and divide operations as long latency instructions and the rest, including data effective address calculation, are short latency operations.

In this paper we proposed a *Ditto Processor* to combine the advantages of two previous schemes and still be able to reduce the performance loss needed for reliable computing. It achieves the goal by handling short and long latency instructions in slightly different ways. After the instructions are decoded, long latency instructions would speculatively execute twice and the results are compared before instructions committed. All instructions are cloned when they are ready to retire. The duplicated instructions are held in a buffer and send back to the beginning of the pipeline. Since results of long latency instructions are checked, the clones of these instructions would not pass execution stage again after renaming operation are verified. For the clones of short latency operations, once they completed the re-execution, results are compared with the results of original instructions. If the results of any types instructions do not match with their clones, processor rollbacks to the point prior to the execution of these instructions.

This approach is unique in several ways:

1. It does not require SMT support and the operation system needs not to be aware of the duplicated instructions.
2. The entire pipeline except the commit stage is covered instead of just functional units. Commit stage must be duplicated in order to have full coverage.
3. Detecting the transient fault of short and long latency instructions in different ways and having fewer penalty cycles for fault recovery help to reduce performance loss. Our simulation result shows 1.8~13.3% performance degradation.

This paper attempts to quantify and compare performance degradation of various time redundant schemes using a microarchitecture simulator when faults are present. It assumes transient faults are few and occur only as isolated single event. When a fault occurs during the simulation, it is always detected. Performance lost due to re-execution and accounting is logged. This paper does not guarantee schemes used will detect all faults. It is organized as follows. In the next section we provide background research in this area. In section 3 we discuss the details of Ditto Processor operation and additional hardware overhead required. In section 4 we describe the simulation experimental setup. In section 5 we present results of our simulation study together with some observations. Finally, we draw some conclusion.

2. Background and Previous Works

Present single-chip Commercial-Off-The-Shelf (COTS) microprocessors have concentrated the design effort on performance. Reliability has not been the primary focus. However, some fault tolerant features have been added into COTS microprocessors [19][31].

Hardware redundancy is one possible approach to cover logic errors. The Pentium® Pro processor family has built-in mechanism to connect two processors into the master/checker duplexing configuration for functional redundancy checking. It allows duplicated chips to compare their outputs and detect errors. However, this technique required 100% or more logic overhead. Other hardware redundancy approaches adopted involve duplicating selected logic within the chip and include error-checking logic in all functional elements. IBM's G5 processor is a good example of this approach [19][21]. G5 duplicates its I-unit and E-unit. It incurs no delay penalty with the duplication because it is able to hide the compare-and-detect cycle completely. Therefore G5 achieves improved checking without any performance penalties. However, there is a 35% circuit overhead.

Recently there has been a resurgence of interest in utilizing time redundancy at the microarchitecture level to recover transient faults. We may classify these related works into two categories. The first category utilizes SMT mechanism to execute two redundant threads in a processor with SMT support. The second type focuses on modifying superscalar processor. We group several existing designs into these two time-redundant schemes.

A. Utilizing SMT mechanism in a SMT processor:

1. Active-stream/Redundant-stream Simultaneous Multithreading (AR-SMT) proposed by Rotenberg [22] exploits several recent microarchitectural trends to protect computation from transient faults and some restricted permanent faults. In this approach, a SMT processor executes an instruction stream called active stream (A-stream) first. Results committed from this instruction stream are stored in a delay buffer. A second stream (R-stream) of instructions tails behind the A-stream with a distance equals to the length of the delay buffer. Results from the R-stream execution are compared with results stored in a delay buffer and committed if they match. Since there are two threads being executed, there are two memory images maintained.
2. Recently, the same research group has proposed a new paradigm for increasing both performance and fault tolerance coverage called "slipstream". Instead of executing two exact instruction streams as in AR-SMT, slipstream processors' A-stream is shortened by the removal of ineffectual instructions. This approach [23] allows the A-stream to run ahead of the R-stream and thus provides not only fault-tolerant coverage but also performance improvement.

3. Work done by Reinhardt and Mukerjee on the Simultaneous and Redundantly Threaded (SRT) processor [33] also utilizes redundant thread in a SMT processor to detect faults. The SRT dynamically schedules the redundant thread to hardware resources to have higher performance. Their work introduces the abstraction called sphere of replication to identify the fault coverage. .

4. Rashid et. al. proposed fault tolerant mechanism in the Multiscalar Architecture [24]. Multiscalar processor usually has many processing units to exploit the instruction level parallelism (ILP). This technique utilizes a minor part of the processing units for re-executing the committed instructions. Both permanent and transient faults in the processor units can be detected.

B. Modified superscalar processor

1. Work done by Franklin [25] utilizes spare resources in a superscalar processor to implement time-redundancy. This approach duplicates all instructions at either the dispatch or the issue stage. Duplicated instructions occupy the otherwise under-utilized functional units to produce checking results for verification.

2. Nickel et. al. [26] extended Franklin's work and tried to improve performance of time-redundant processors by adding spare capacity. After an instruction completes execution but before it is retired, a duplicated copy is placed in a FIFO queue. This duplicated instruction is re-scheduled and re-executed. In order to minimize the performance loss, this method also strategically adds extra functional units to the pipeline.

3. Ray et. al. [32] proposed a similar scheme to what Franklin has done. A single instructions stream creates multiple redundant threads at decode stage and results from duplicated threads are verified at commit stage.

4. Mendelson et. al. [27] mentioned that if the decoding logic is not implemented by table lookup (memory structure) one needs to employ some methods to protect it from transient errors also. However, their approach focused on re-executing the operations twice at execution stage and verifying results before instruction commit. This scheme has minimum hardware requirement to perform error checking and has less performance impact due to error detection. However, compare to previous studies, this scheme has less fault coverage in that it only verified the correctness of functional units.

5. Austin et. al. [34][35][36] introduced the concept of using a less complex checker named DIVA to verify faults. The DIVA checker can verify not only transient faults but also design faults. Moreover, the performance impact of this extra checking mechanism is less than 3%.

In summary, we found works on using SMT to detect fault have better fault coverage but suffer higher percentage performance loss. While works on using existing superscalar processor, they do not cover the fault that may occur at the frond-end of the pipeline. In this study, our goal is to provide a fault tolerant processor which has low cost, low performance degradation and high fault coverage. We use a microarchitecture simulator to quantify the performance loss of several schemes.

3. Design of Ditto Processor

Ditto Processor differs from previous approaches in that it splits long latency operations and short latency operations into different verification path. After the instructions are decoded, long latency instructions are identified and speculatively executed twice. Results of these long latency instructions are compared but they are not committed. All non-speculative instructions including those non-speculative long latency instructions are cloned before retirement. These duplicated instructions are held in a buffer and send back to the beginning of the pipeline. Since the result of long latency instructions are executed twice and checked, the clones of these instructions would not pass execution stage again after renaming operation are verified.

For the clones of short latency operations, once they completed the re-execution, results are compared with the results of original instructions. Any transient fault can potentially be discovered when the result of the re-execution differs from that of the original execution and simple recovery scheme is used. If faults occurs at the decode stage, results will differ also and be detected.

Prior to our main study, we observe, in an average, only 12% of the resources are utilized for integer and floating-point applications on a baseline 8-issue superscalar processor. This means that there are plenty of opportunities to take advantage of these unused resources to hide the overhead of program re-execution, verification and transient fault recovery. However execution in a superscalar tends to be busty at times. Without careful organization, time-redundancy through cloning still degrades its performance.

In the following sections we will describe in more details the design of Ditto processor. After reading through the design details, interested reader may find an example of pipeline flow for a small piece of sample code in the appendix.

3.1 What hardware is added to support fault-tolerant mechanism?

Figure 1 illustrates the basic microarchitecture diagram of *Ditto processor.* It has two additional blocks - a "delay buffer" and a "verify logic". Several existing blocks in a superscalar processor also need to be modified. These include the re-order buffer (ROB), the commit logic, the fetch unit and the decode unit. We describe the changes needed for each of these blocks.

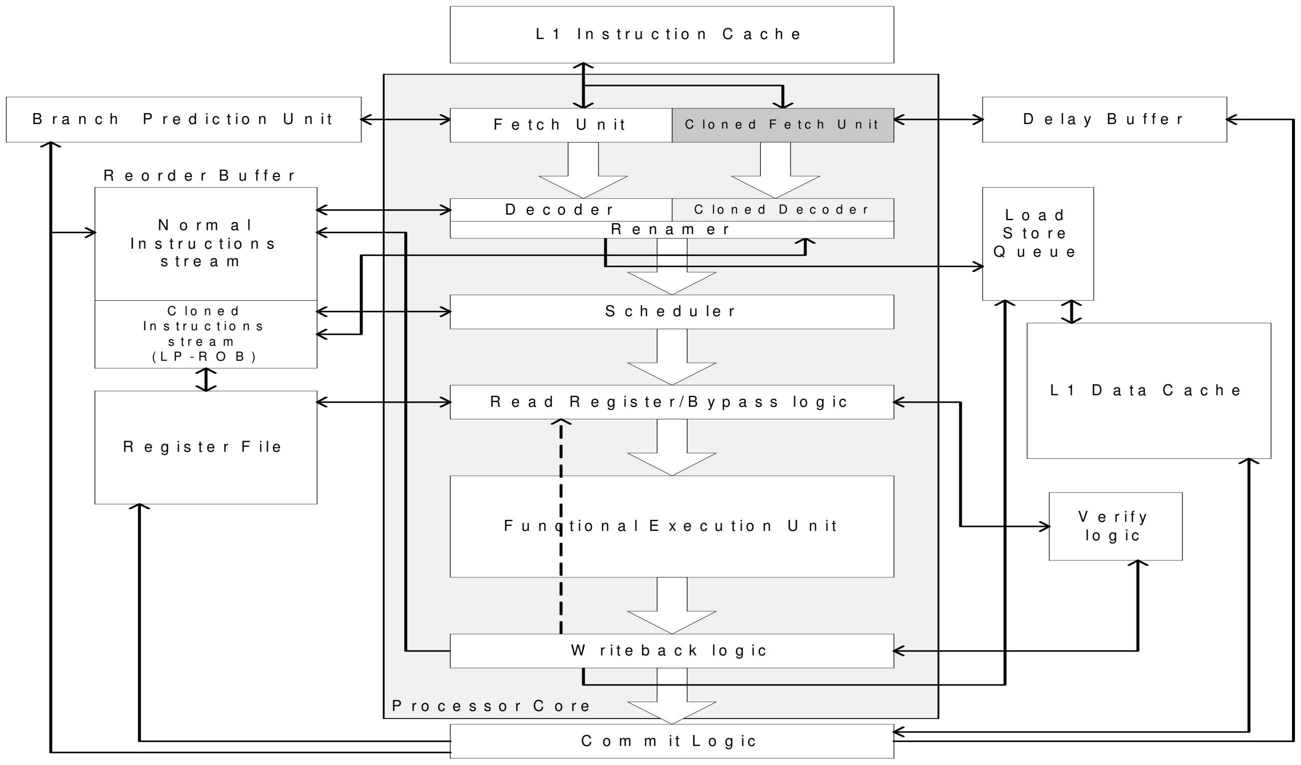


Figure 1 Basic Architecture of *Ditto Processor*

Delay Buffer: Instructions are executed normally the first time. Results of committed instructions are queued in the delay buffer similar to other schemes [22][23][26]. However, each entry not only stores the result but also includes the associated instruction code and its instruction address. For long latency operations, we also allocate the immediate entry that follows to store source operands' values. We called these instructions stored in the delay buffer *cloned instructions*. These cloned instructions are removed from the delay buffer when they are scheduled and passed the registered read stage.

Fetch and decode units: Since the gap between processor cycle time and memory access time will likely grow wider each year, most likely fetch and decode units are not the bottleneck. We choose to split the fetch and decode units into two equal parts. Half of the fetch and decode unit is reserved for cloned instructions stream. In order to simplify the maintenance of normal instructions and cloned instructions stream, an extra program counter is added for the cloned instructions stream.

Reorder Buffer: We also found that the average reorder buffer (ROB) occupancy in the baseline non-fault-tolerant system with 128-entries ROB is about 50% for integer benchmark and 90% for floating point benchmark. By allocating the redundant part of ROB to cloned instructions stream, we may reduce the performance degradation without extra hardware overhead. After the cloned instructions are decoded, they are placed at the lower part of ROB (LP-ROB) as illustrated in Figure 1. Results of normal instructions are copied from the delay buffer to the result field of LP-ROB. Error Correction Code (ECC) checking mechanism protects this copy operation. In order to differentiate long latency instruction and short latency instruction, extra bit is added to each ROB entry. We will describe how to handle cloned instructions stream renaming in section 3.3. Furthermore, the size of LP-ROB should be small enough to minimize the effect of normal instructions stream's throughput. Our study reveals that long latency operations would have severe impact on LP-ROB pressure and degrade the performance accordingly. Hence, we suggest that short and long latency instructions should go through different verification path.

Status bit to handle duplicate execution: Since all long latency instructions are executed twice including those that are speculative, we adopt the idea from [27] to handle these duplicate computations. This approach requires the fewest hardware overhead. An extra status bit is added to each of the ROB entries indicating the long latency operation is ready to be executed the second time. Since memory reference micro-ops belong to long latency operations, this extra status bit is also appended to entries of the load store queue (LSQ). Furthermore, the verify bit is used to confirm that the computation of duplicated long latency operations were completed and verified. After these results are confirmed, results from duplicated copies are discarded. Since results from the original instruction and the duplicate copy may be ready at different cycles, we also need to address the scheduling of their dependent instructions. We schedule dependent instructions according to the data ready time of the original copy, since faults are not as frequent This cause no further complication because a mismatch of results will bring back execution prior to the faulty instruction.

Verify logic: Once these cloned instructions complete their execution, cloned instructions' results are compared to the original instructions' results saved in the result field of ROB. Verify logic, next to the write-back stage, is used to handle this error detection and recovery. We will present this mechanism in the following section.

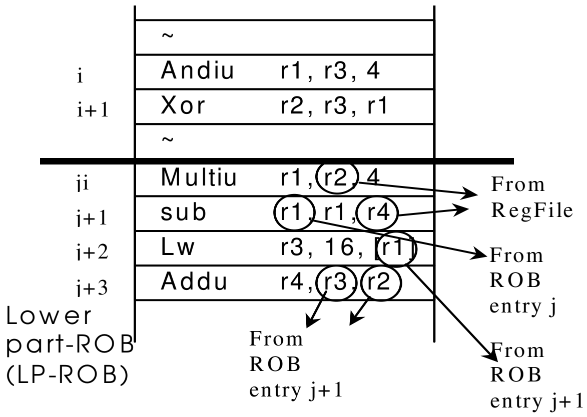
3.2 Error detection and Fault recovery mechanism

Ditto Processor employs two checking mechanisms to detect potential transient faults. The first mechanism is placed after the register-read stage. After a cloned instruction's source operands are ready, we compare the decoded instruction with the correspond entry in the delay buffer. It detects two places where transient faults may occur.

1. If this re-fetched instruction does not match the correspond entry in delay buffer, it indicates the occurrences of a transient error in the fetch unit or in the decoder. For conditional and unconditional jump instructions, the decoded target address is also verified.
2. For long latency operations if clones source operands values does not match the correspond values in the delay buffer, it indicates the occurrences of a transient error in renaming logic.

This mechanism allows us to detect faults occurs at earlier stages of the pipeline. The verification process is overlapped with the execution stage and poses no extra delay.

Figure 2 Instruction renaming example



The second checking mechanism occurs when the cloned instructions complete their computation. Results of cloned instructions are compared to the original results stored in the result field of ROB entries. If the results are the same, cloned instructions are removed from reorder buffer. If results do not match, then we have detected a transient fault in functional units. Since long latency instructions already verified computation results while in normal instructions stream, these instructions would not go though this second mechanism.

In both mechanisms, we recover the system back to the known correct state similar to branch mis-prediction recovery. Hence, there is no other extra hardware needed beside what we have mentioned to handle this error recovery on register file rollback. We will present this rollback mechanism in the following section. Since, in this study, we assume the mean time between faults (MTBF) is about 10 million cycles, after several cycles of error recovery, the second try[1] should have a valid result and program may continue to execute.

3.3 Cloned instruction renaming and register file rollback mechanism

Since the decoder of normal instructions stream and cloned instructions stream come from different paths, the renamer should not mix these two streams together. Figure 2 shows a snapshot of the ROB during execution. *Ditto processor*'s ROB is divided into two regions – the normal ROB entries region and the LP-ROB entries region. The LP-ROB maintains the program order of cloned instructions stream while the rest of the ROB is used for normal instructions stream.

We present an example to describe how *Ditto Processor* handles instructions renaming. Let's assume the LP-ROB starts with entry *j*. Since "multiu" is at the head of LP-ROB, all previous cloned instructions have been verified. The source operand (r2) of "multiu" is mapped to architecture register file, so is the source operand (r4) of "sub" and (r2) of "addu". The source operand "r1" of instruction "sub" is depending on the previous result of entry *j*. Since the previous result has been copied from delay buffer to entry *j* as described in section 3.1, the source operand (r1) of "sub" may use this value and schedule immediately after renaming. This is true for instructions "lw" and "addu" also. This scenario contains no data hazard and allows cloned instructions to fly through pipeline stages faster then normal instructions. It also reduces possible performance loss due to re-execution come with the time-redundant technique.

For long latency operations, if transient error occurs in this renaming operation, the verify logic will detect the source operands' values are different from values produced by the original instruction and will signal the recovery mechanism. For short latency operations, the verify logic would detect this renaming error if the clone's computation result is different from the original result since clone instruction stream and normal instruction stream handle renaming operation independently.

In a redundant processor using simultaneous multithreading technique such as AR-SMT, each thread must maintain its own register status and values by register map [22], it requires some additional hardware when compared with *Ditto Processor*. In *Ditto Processor* we only need to augment the state bits in architecture register file.

[1] The second try means the instructions will be fetch, decode and execute twice as mentioned and the result will be verify again.

Whenever a normal instruction is ready to commit, it writes the result to register file and transits the status bits from "invalid" to "transient". Once the cloned instruction is verified, the status is changed from "transient" to "verified". This approach requires only one extra bit added to each register. From the re-namer and scheduler's point of view, they treat "transient" and "verified" value in the same way as data ready. If a transient error is detected, all "transient" values are flushed from the architecture register file. Moreover, all in-fly instructions are squashed similar to miss-branch prediction recovery.

3.4 What types of operation are protected?

In the *Ditto Processor* design, we cover every type of instructions for possible transient error. However, we do assume that there is no self-modifying instruction in our system.

Short latency Arithmetic/logic instructions: After these instructions are ready to retire, they store the result and other information to the delay buffer and ROB entry is free for other normal instructions. The cloned instruction is then fetched, decoded/renamed, scheduled and executed. After the result is verified, the LP-ROB is free for other cloned instructions. Since we assume the Branch Prediction Unit is protected by the ECC mechanism, our scheme may verify the correctness of decoded target address and the outcome of branch.

Multiply/Division instructions: Since these instructions have long execution latency, they are duplicated after decode and speculatively execute twice and result are compared and verified. Result of these instructions and other information are stored in delay buffer for verification later. These instructions are also cloned and re-fetched. However, after it is decoded/renamed, scheduled and read from register, they would not go through computation again. As mentioned before they are checked by the first checking mechanism. After passing the first checking mechanism, these instructions are free from LP-ROB.

LOAD/STORE type instructions: After this type instruction was decoded, it generated two micro-ops: one for data address calculation and the other one for memory reference. Since memory micro-op belongs to long latency operation, it would access cache memory twice based on the normal instruction's calculated data address. When this type instruction is ready to commit, it would store the result and other information into delay buffer. After the clone instruction is decoded, it would discard the memory micro-op since we only need to verify the correctness of data address.

3.5 What are protected units?

From Figure 1 we see that processor core is inside the shaded area. In other words, we assume any units outside of this area are protected by ECC logic. Furthermore, any wires and control signals that communicate between processor core and other units, such as data cache or ROB, are also protected by other fault-tolerant techniques [3-6][10-11][14]. Whenever a system interrupt or exception occurs, protection logic will guard the transient fault to make sure these requested are being served correctly. Since the correctness of commit logic is imperatively important on placing the result into delay buffer and this logic is very small, we duplicated the commit logic to enforce its correctness.

4. Simulation Configuration

We modified the SimpleScalar simulator [28] in order to evaluate the performance degradation of different redundant schemes when transient faults are present. We randomly generate faults with MTTF of 10 millions cycles. When each fault occurs, it could occur at any point of the pipeline. In our study we randomly assign the fault to a particular pipeline stage. 14 SPEC2000 benchmarks (8 integers, 6 floating points) [29] are used for our simulation study. All benchmarks are executed for 500 million committed instructions after skipping the first 500 million instructions.

4.1 Baseline Model

In our baseline model, we extend the existing SimpleScalar pipeline model into seven stages: fetch, decode/rename, schedule, register read, execution, writeback and commit. Each stage takes one cycle. In order to eliminate the effect of data speculation, we schedule the dependent instruction at the data ready cycles. For example, in a cache-hit case, load operation takes 3 cycles to access data (2 cycles to access the tag array to determine hit/miss and 1 cycle to access data array). The load dependent instructions will be scheduled 2 cycles later after data effective address is calculated. Table 1 shows the overall baseline system parameters.

Fetch, decode, issue, commit width	8
Branch Predictor Branch Target Buffer	Gshare, 64-entry, 8 way, 8k-entry, 8 way
ROB / LSQ size	128/128 entries
L1 I/D cache	16KB/16KB 4-way, 32B line size
L1 I/D cache hit latency	1/3 cycles
L2 cache	1MB size 8-way, 32B line size
L2 / Memory latency	10/100 cycles
# of pipelined integer ALU/MULT/DIV	4/1/1
Integer ALU/MULT/DIV latency	1/3/20
# of pipelined floating point Adder/MULT/DIV	4/1/1
Floating point Adder/MULT/DIV latency	2/4/24
Read/Write port	4

Table 1 Baseline model system parameters

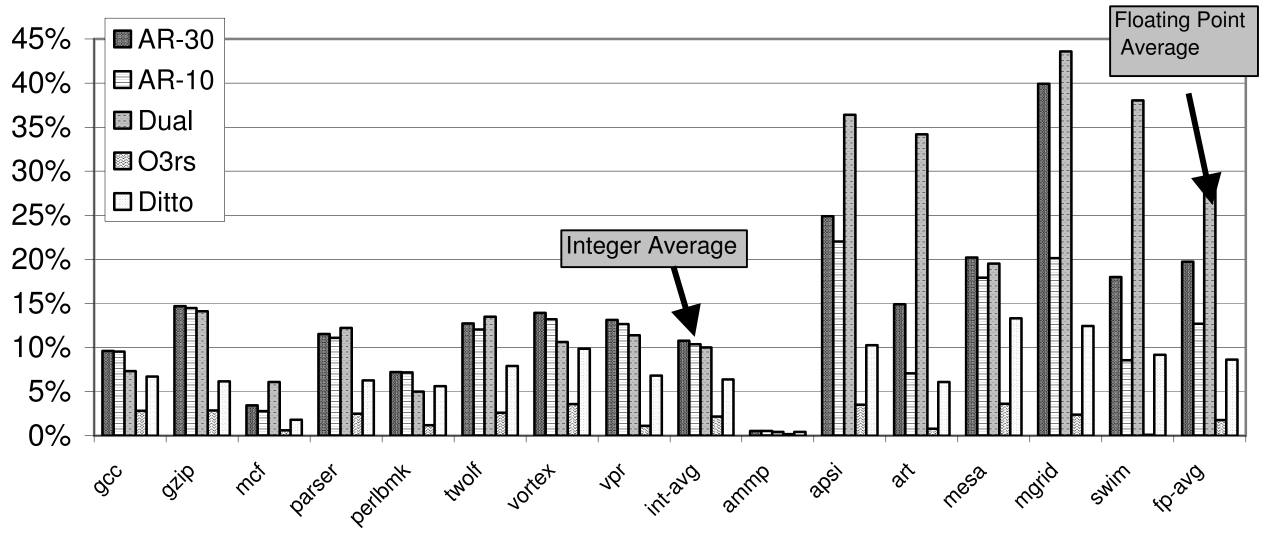


Figure 3 Instruction Per Cycle (IPC) Degradation

4.2 Different Simulation Machine Model

In this study, we compare five different machine models. The baseline model (Base) is described in section 4.1 In order to compare *Ditto Processor* with AR-SMT [22], we adapt AR-SMT into superscalar model, named AR model. There are two differences between AR-SMT and AR model. First, R-stream in AR model does not perform memory reference micro-ops because this operation would require operation system to be aware of A-stream and R-stream. Second, AR model does not contain trace cache. We further define that 30% of ROB entry allocated to R-stream would be AR-30 model (96 entries for A-stream and 32 entries for R-stream). Similarly, R-stream of AR-10 model would utilize 10% of ROB entry (Similarly, 112 and 16 entries).

In our experiment, *Ditto Processor* model's (Ditto) cloned instructions utilize 10% of ROB (16 entries for LP-ROB). Our study reveals that this allocation strategy would have the least performance effect on normal instructions stream. Both AR and Ditto model use 128-entry delay buffer to store the committed instructions.

We also model the 2-way redundant scheme (Dual) by Ray et. al. [32]. The Dual model has the same system parameters as our Baseline. There are two differences between Ray's original architecture and ours Dual model. First of all, the original design has 64KB I-cache, 32KB D-cache, 512KB L2 cache and 2 read/write ports while our memory subsystem modeling is slighted different and summarized in Table 1. Second, ours model has longer pipeline stages. Despite these differences, our dual model matches their result closely.

Out-of-order Reliable Superscalar (O3rs) [27] is also implemented in this study for comparison. System parameters of O3rs are the same as the Baseline model. The O3rs model should have the best result in terms of Instructions Per Cycle (IPC) degradation, since it only verifies the functional units and it does not take away ROB entries from normal instructions like other schemes for re-computation.

4.3 Fault Injection Mechanism

In our study, we inject faults randomly at different stages every 10 million cycles for all different schemes described above. In other words, the fault could be at fetch unit, decoder, scheduler, register read operation, execution, bypass logic or others. As instructions with faults pass through our checking mechanism they will be detected as described in section 3.2. Machine will be reset back to the known state. In this study, we assume two cycles of error detection and recovery penalty.

5. Performance result

In this section, we present the simulation result of our study. Section 5.1 shows the IPC degradation of each model and section 5.2 shows the functional units resource utilizations of each model. Section 5.3 presents the characteristic of *Ditto Processor.*

5.1 Performance degradation

Figure 3 illustrates the percent performance degradation of several time-redundant fault-tolerant designs. AR-10 has slight performance improvement over AR-30 in that the former utilizes less LP-ROB. However, for one of the floating-point benchmark - "mgrid", it shows a large difference in performance. Further study reveals that "mgrid" has over 65% of long latency instructions. As LP-ROB size reduces, it leaves more space in upper ROB to occupy long latency instructions and, in turns, reduce the performance loss. The average floating-point benchmark result also shows the same behavior that AR-10 outperforms AR-30 by about 7%.

Since integer benchmarks have over 70% of short latency operations* and these operations enter and leave LP-ROB within a very short time, they give AR-10 only slight advantage over AR-30.

In the Dual model, after the instructions are decoded, it created another copy of all instructions. A duplicated instruction also occupies a ROB entry as described in [32]. This mechanism reduces the effective size of ROB by half. Therefore, this scheme suffers severe performance loss in floating-point benchmarks and “mcf”. In these cases, compared to AR-30, Dual model degrades the performance by about 9% in floating point benchmarks and 3% in “mcf”.. The O3rs model has the least performance loss among five models because it does not take away ROB entries for duplicated instructions. O3rs loses 1.7% and 2% performance for integer and floating-point benchmarks respectively. As mentioned O3rs does not cover front-end part of the pipeline nor memory instructions. Since in our Ditto model the cloned long latency instructions do not pass through execution stage again and reduce the pressure on LP-ROB, this further reduces the performance loss. Ditto suffers about 1.8~13.3% performance degradation.

We also observe that “ammp” benchmark has very little performance loss, only about 0.4%, on all models. Further study reveals that “ammp” has very high L1 and L2 data cache local miss ratio, about 50% and 90% respectively, most of the operations are hinder by lengthy memory reference. In this case all our simulated fault-tolerant models may be able to benefit from normal instructions stream's low throughput and low functional units utilization.

In summary, AR-30, AR-10 and Dual model has an average of 10% performance degradation on integer benchmarks. Ditto model outperforms these three models and reduces the performance loss by 40% to about 6% on integer benchmarks. For floating-point benchmarks, the performance loss of AR-30, AR-10 and Dual models are about 19%, 12% and 28%, respectively. Ditto reduces the degradation by 30% and 70% respectively to 8.6% when it is compared with AR-10 and Dual models on floating-point benchmarks.

5.2 Functional unites resource utilization

Since different models have different effects on functional unit’s resource utilization rate, Figure 4 presents each model’s utilization ratio in more detail. Compared to the Base model, all other models have better functional unites utilization, especially Ditto. Since Ditto model verifies all types of instructions, it utilizes resource more efficiently. On average, Ditto utilizes integer ALU units about 15% more than Base model.

Dual model has the similar ratio as Ditto in integer benchmarks, but since it duplicates all instructions including instructions that are speculative, the performance loss is higher. For floating-points benchmarks, there are more long latency instructions putting more pressure on the ROB. This further reduces the effective instructions windows size. For example, Dual model only uses half of the ROB to explore instructions level parallelism (ILP). Hence, Dual has the worse integer ALU utilization rate for floating-point benchmarks. Because both Ditto and Dual models verify cache access operations, memory ports are used more efficiently on these two models.

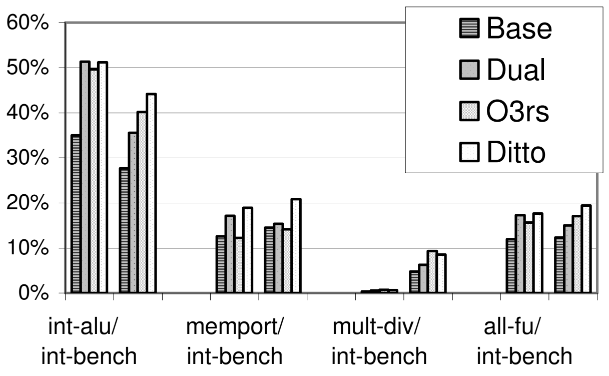


Figure 4 Average functional units resource utilization[2]

O3rs posts no effect on cache ports since O3rs does not verify memory reference micro-ops. Since integer benchmarks rarely use multiplier and divider, all models has very little utilization rate for these modules. O3rs model has slight better MULT/DIV unit utilization for floating-point benchmarks in that it has more ROB entry to explore ILP than Ditto[3]. In summary, we observe that, when compared to Base model, Dual model has about 5% more on functional utilization for integer benchmarks and 3% more for floating-point benchmarks. O3rs model is 3.6% and 4.7%, respectively. For Ditto model, it is 5.7% and 7.4% better on average. Hence, Ditto model has full transient fault coverage with less performance degradation.

5.3 The characteristic of Ditto Processor

Figure 5 depicts the percentage of IPC degradation when we compare the Ditto to the Base model with different L1 cache hit latency. We observe that as L1 cache hit latency increases, the Ditto model gradually reduces the performance loss on both integer and floating-point benchmarks. One factor that affects the percentage of performance degradation is the amount of idle time available in the processor for time redundancy to perform transient fault checking. The basic motivation for our approach is to utilizing these stalled processor cycles to

[2] There are four groups in this figure and each group contains integer and floating-point benchmarks result. The most left group is integer ALU unit utilization ratio. The second group is memory port utilization ratio and the third group is the combination of integer and floating-point multiplier/Divider unit utilization ratio. The most right side group is overall functional unit utilization ratio.

[3] Ditto model has 112 ROB entries for normal instructions stream and 16 entries for cloned instructions stream.

verify computation through re-execution. As memory latency increases in terms of cycle time, we have more stalled cycles in the processor and more resources available. This gives more opportunity to perform cloned instructions execution and reduce the effect of performance degradation.

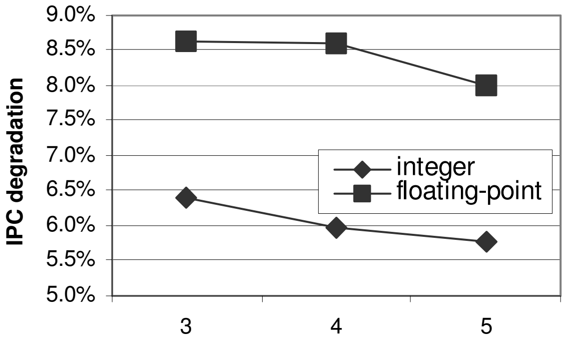


Figure 5 Effect of L1 cache hit latency on Ditto

6. Conclusion

In this paper we have presented the detail design of a fault tolerant superscalar processor called *Ditto Processor*. This processor re-fetches and re-decodes all instructions to protect all pipeline stages' logic from soft errors to assure high computation confidence. It requires little extra hardware on top of the baseline superscalar. We explain the additional microarchitecture resources needed and what units we can protect. We also explain how to handle the register renaming in *Ditto Processor*. We further identified that long latency operations have significant impact on time-redundant fault-tolerant superscalar processor. We studied the performance degradation of *Ditto Processor* in comparison with baseline superscalar and other published schemes. In general, *Ditto Processor* suffers only 1.8~13.3% of performance degradation for all benchmarks.

As *Ditto Processor* have only 1~6% more performance loss compared to O3RS scheme, our scheme have much better fault coverage. The degree of reduction varies with amount of contention on the resources brought about by duplication. We also observed that as memory latency increases, the performance degradation on *Ditto Processor* is reduced. While memory processor performance gap continues to grow with technology advancement, there will be more stalled cycles available for time redundancy. Our study reveals that different applications have different characteristics and have various requirements on hardware resources. Adopting the time-redundant fault-tolerant technique based on this knowledge would provide a balance designed fault-tolerant computing environment with less performance loss.

Acknowledgement

Authors want to thank anonymous reviewers for very helpful comments.

7. References

[1] T. Juhnke and H. Klar, "Calculation of the Soft Error Rate of Submicron CMOS Logic Cicuits," IEEE JSSC, Vol. 30, No. 7, July 1995, pp. 830-834.

[2] J. Robertson, "Alpha Particles Worry IC Makers as Device Features Keep Shrinking," Semicond. Business News, October 21, 1998.

[3] N. Cohen et. al., "Soft Error Considerations for Deep-Submicron CMOS Circuit Applications," Proc. of IEDM, 1999, pp. 315-318.

[4] P. Hazucha and C. Svensson, "Impact of CMOS Technology Scaling on the Atmospheric Neutron Soft Error Rate," IEEE Trans. On Nuclear Science, Vol. 47, No. 6, Dec. 2000, pp. 2586-2594.

[5] T. Karnik et. al., "Scaling Trends of Cosmic Rays Induced Soft Errors in Static Latches Beyond 0.18um," Symposium on VLSI Circuits Design of Tech. Papers, 2001, pp. 61-62.

[6] U. Gunneflo, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation," Digest of Papers in the 19th International Symposium on Fault-Tolerant Computing, 1989, pp. 340-347.

[7] G. Miremadi, and J. Torin, "Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection," IEEE Transactions on Reliability, Volume: 44, Issue: 3 , Sept. 1995, pp. 441 –454.

[8] R. Horst et. al., "The Risk of Data Corruption in Microprocessor-based Systems," Digest of Papers in the 23rd International Symposium on Fault-Tolerant Computing, Aug. 1993, pp. 576 –585.

[9] Barry W. Johnson, Design and Analysis of Fault Tolerant Digital Systems, Addison-Wesley, 1989.

[10] A. Avizienis, "Toward Systematic Design of Fault-Tolerant Systems," IEEE Computer, April 1997, pp. 51-58.

[11] D. K. Pradhan , Fault-tolerant computer system design, Prentice-Hall , 1996.

[12] W. Torres-Pomales, "Software Tolerance: A Tutorial," NASA Tech. Memorandum, TM-2000-210616, Langley Res. Center, Hampton Virginia, Oct. 2000.

[13] R.E. Blahut, Theory and Practice of Data Transmission Codes, Addison-Wesley, 1983.

[14] Parag K. Lala, Self-Checking and Fault-Tolerant Digital Design, Academic Press, 2001.

[15] G. Sohi, "A Study of Time-Redundant Fault Tolerance Techniques for High-performance Pipelined Computers," Digest of Papers in the 19th International Symposium on Fault-Tolerant Computing, 1989

[16] J. H. Patel and L. Y. Fung, "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands," IEEE Trans. On Computers, Vol. C-13, No. 7, July 1982, pp. 581-595.

[17] M. Nicolaidis, "Time redundancy based soft-error tolerance to rescue nanometer technologies," Proceedings of 17th IEEE VLSI Test Symposium, 1999, pp. 86 –94.

[18] M. Franklin, "Incorporating Fault Tolerance in Superscalar Processors," Proceedings of 3rd International Conference on High Performance Computing, 1996, pp. 301 –306.

[19] A. Avizienis and Y. He, Microprocessor entomology: a taxonomy of design faults in COTS microprocessors, Dependable Computing for Critical Applications 7, 1999, pp. 3 –23.

[20] L. Spainhower and T. A. Gregg , "IBM S/390 Parallel Enterprise Server G5 fault tolerance: A historical perspective," IBM J. of Research and Development, Vol. 43, No. 5/6, 1999.

[21] Y. He et .al ."Assessment of the applicability of COTS microprocessors in high-confidence computing systems: a case study," Proceedings International Conference on Dependable Systems and Networks, 2000

[22] E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors," Digest of Papers in 29th International Symposium on Fault-Tolerant Computing. 1999, pp. 84 –91.

[23] K. Sundaramoorthy et. Al. "Slipstream Processors: Improving both Performance and Fault Tolerance," 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems, November 2000.

[24] F. Rashid et. al., "Fault Tolerance Through Re-execution in Multiscalar Architecture," Proc. Conference on Dependable Systems and Networks, 2000, pp. 482 –491.

[25] M. Franklin, "A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors," Digest of Papers in the 25th International Symposium on Fault-Tolerant Computing, Aug. 1995, pp. 207–215.

[26] Joel Nickel, "REESE: A Method of Soft Error Detection in Microprocessors," M. S. Thesis, Dept. of ECE, Iowa State University, Ames Iowa, 2000.

[27] A. Mendelson and N. Suri, "Designing High-Performance and Reliable Superscalar Architectures The Out of Order Reliable Superscalar(O3RS) Approach," Proc. Conference on Dependable Systems and Networks, 2000,

[28] Doug burger, Todd M. Austin "Simplescalar Tool Set Version 2.0," Technical Report #1342, June 1997,University of Wisconsin-Madison Computer Science Department

[29] http://www.spec.org/osg/cpu2000/docs/readme1st.txt

[30] S. Sair and M. Charney, "Memory Behavior of the SPEC2000 Benchmark Suite," IBM Research Report, RC 21852 (98345), Oct. 6, 2000.

[31] A. Avizienis, "A Fault Tolerance Infrastructure for dependable Computing with High-Performance COTS Components", In proceeding of Dependable Systems and Networks, 2000.

[32] Joydeep Ray, et. al., "Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery", In proceeding of 34th Microarchitecture, December,2001.

[33] Steven k. Reinhardt et. al. "Transient Fault Detection via Simultaneous Multithreading", In proceedings of the 27th International Symposium on Computer Architecture, June, 2000

[34] T. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design", In proceeding of 32nd Microarchitecture, November, 1999.

[35] T. Austin, "DIVA: A Dynamic Approach to Microprocessor Verification", The Journal of Instruction-Level Parallelism Volume 2, 2000.

[36] Saugata, et. al. "Effective Checker Processor Design", in proceeding of 33rd Microarchitecture, December 2000

[37] A Yoaz et. al. "Speculation Techniques for improving load related Instruction scheduling", Proc. Of 26th Int'l Symp. On Computer Architecture, May, 1999.

APPENDIX: Examples of pipeline flow and fault detection

A. Sample program instructions:

Instruction	Label in the pipeline diagram
next_loc:Multiu r1,r2,4	A
Sub r1,r1,r4	B
Lw r3,16[r1]	C
Addu r4,r3,r2	D
Bne r4,r5, [next_loc]	E

B. Pipeline diagram:

ROB	FET	Dec	SCH	REG	EXE/MEM	WB	CMT
1	A1[4]	A2	A3	A4	A5-7	A8(CHK)	A9
2	B1	B2	B8	B9	B10	B11	B12
3	C1	C2	C11	C12	C13-16	C17(CHK)	C18
4	D1	D2	D17	D18	D19	D20	D21
5	E2	E3	E20	E21	E22	E23	E24
6	A'3	A'4	A'5	A'6	A'7-9	A'10(CHK)	A'11
7	B'3	B'4	B'20	B'21	B'22	B'23	B'24
8	C'3	C'4	C'23	C'24	C'25-28	C'29(CHK)	C'30
9	D'3	D'4	D'29	D'30	D'31	D'32	D'33
LP-ROB	~	~	~	~	~	~	~
1	$A10	$A11	$A12	$A13	$A14(CHK)		
2	$A'12	$A'13	$A'14	$A'15	$A'16(CHK)		
3	$B13	$B14	$B15	$B16	$B17(CHK)	$B18(CHK)	
4	$C19	$C20	$C21	$C22	$C23(CHK)	$C24(CHK)	
5	$D22	$D23	$D24	$D25	$D26(CLK)	$D27(CLK)	
6	$E25	$E26	$E27	$E28	$E29(CHK)	$E30(CHK)	
7	$B'25	$B'26	$B'27	$B'28	$B'29(CHK)	$B'30(CHK)	
8	$C'31	$C'32	$C'33	$C'34	$C'35(CHK)	$C'36(CHK)	
9	$D'34	$D'35	$D'36	$D'37	$D'38(CHK)	$D'39(CHK)	

C. Examples of fault detection:

Case1: A fault occurs at the FET stage of inst. B would be detected at $B17. Inst. C, D, E, B', C', D' would be squashed from the ROB. Inst. $B would be squashed from the LP-ROB

Case 2: A fault occurs at the DEC stage of inst. E would be detected at $E29. Inst. E, C', D' would be squashed from the ROB. Inst. $E, $B' would be squashed from the LP-ROB.

Case 3: A fault occurs at the EXE stage of inst. A' would be detected at A'10. Inst. B, C, D, E, A', B', C', D' would be squashed from the ROB.

Case 4: A fault occurs at the WB stage of inst B would be detected at $C23. Error will occur in propagating the result to B's dependent instructions. Inst. E, C',D' would be squashed from the ROB. Inst. $C, $D would be squashed from the LP-ROB.

Case 5: A fault occurs at the REG stage of inst A' would be detected at $A'16. Inst. B', C', D' would be squashed from the ROB. Inst. $A' would be squashed from the LP-ROB.

[4] The number following the instruction label indicates the cycle time. An "'" means it is a cloned instruction. For example A'3 means instruction A cloned at cycle time 3.