Designing High-Performance & Reliable Superscalar Architectures The Out of Order Reliable Superscalar (O3RS) Approach[1]

Avi Mendelson
Microprocessor Research Lab,
Intel Corporation, Israel
Avi.Mendelson@intel.com

Neeraj Suri
Dept. of Computer Engg.,
Chalmers University, Sweden
suri@ce.chalmers.se

Abstract

As VLSI geometry continues to shrink and the level of integration increases, it is expected that the probability of faults, particularly transient faults, will increase in future microprocessors. So far, fault tolerance has chiefly been considered for special purpose or safety critical systems, but future technology will likely require integrating fault tolerance techniques into commercial systems. Such systems require low cost solutions that are transparent to the system operation and do not degrade overall performance. This paper introduces a new superscalar architecture, termed as 03RS that aims to incorporate such simple fault tolerance mechanisms as part of the basic architecture.

Keywords: Superscalar architectures, Pipelines, Transient Errors/Recovery

1 Introduction

In order to achieve high integration and low power consumption, the trend in VLSI technology has been to shrink the geometry paralleling Moore's law of doubling device density every 18-24 months. Several companies [1] are currently fabricating chips using 0.18-micron technology, and it is expected that in 3-5 years the technology will reach the 0.1-micron limit. Such advanced technologies present some new challenges and concerns that are not apparent for current technology [6]. Among these new challenges is the need to protect the system from the disruptive effect of transient faults by integrating simple, low-cost fault tolerance mechanisms as part of the basic design for future commercial micro-architectures.

The need to protect against transients by integrating fault tolerance mechanisms arises from varied considerations such as (a) At 0.1 micron technology, the dv/dt ratio and expected transistor's threshold level indicates the growing occurrence of transient faults. A ballpark number is of 1-3 transients/day for an average application operating continuously, (b) It is expected that more and more asynchronous designs and dynamic logic circuits will be used, which are particularly prone to timing and transient faults. Thus, we can expect transients or "infrequent" faults (faults that appears only in small subsets of operational modes) will become part of future processor/computational models.

Traditionally, fault tolerance techniques are designed to protect safety critical systems such as aerospace, nuclear and similar control systems. Such systems often can trade performance and costs for dependability. Providing similar fault coverage to commodity processors is not a simple issue. Here, on one hand, both cost and high performance are essential, and on the other hand, the level of fault-coverage is important as well given their growing and pervasive role in embedded systems.

At the processor/architectural level, a variety of space/time redundancy techniques currently exist for integrating fault tolerance techniques. Though our interest in this paper is on time redundancy, for completeness, we briefly outline other commonly used time and space redundant fault tolerant techniques, namely:

- Error detection/error correction codes (ECC) used widely for protecting memory cells [7].
- Spatial redundancy techniques entailing physically duplicating the system (or specific functional units) so that the same program will be executed by different pieces of hardware in parallel and their execution results could be compared to verify the correctness of the execution. [8]. Such systems are relatively expensive and can be used for custom designs, but not for high performance commodity systems. An alternative to duplicating the entire system was proposed in [9], where instructions that reach the "instruction window", get duplicated and get sent to two independent execution-units. In this

[1] Work supported in part by NSF CAREER Grant CCR-9896321

technique, only the ALU part is duplicated.

- Temporal redundancy techniques propose re-executing each program (or fragments) and to test/compare their outcomes. The potential drawback of this scheme is that it might significantly reduce the overall performance. Recently, [5] proposes a new temporal redundancy based scheme termed AR-SMT that aims to limit the performance loss.

Our focus is on temporal redundancy approaches, and this paper aims to present a new architectural approach that addresses the processes of detection and recovery from transient errors in the logic of the computer, i.e., in the computational elements such as ALU, FPU, and address generation units. Towards these objectives, this paper is organized as follows. Section 2 presents the general structure of current superscalar architectures prior to addressing fault tolerance aspects in them. Section 3 reviews contemporary architectural schemes utilizing temporal redundancy. After introducing our proposed out of order reliable (O3RS) superscalar architectural approach in Section 4, Sections 5 and 6 develop the error recovery approaches. First, we present an approach to basic support for transient fault detection and recovery. Next, enhanced approaches to address transient recovery are developed. Section 7 outlines the performance results for the proposed approaches.

2 Superscalar Processors: Basics

In this section we present the structure of a typical superscalar architecture [2] similar to the Pentium-II family [3]. This section primarily establishes the conceptual structure and superscalar related terminology we will use throughout this paper.

Figure 2.1 depicts the basic structure of the execution pipeline of a general purpose, out-of-order superscalar processor with a reservation station(s). Instructions are fetched from the main memory or from the instruction cache in the program order into the decode unit. The decode unit is responsible for registering the instructions in the ROB (re-order buffer), for renaming their registers (from the user view registers into the the architectural registers) and for sending the instructions to the reservation tables that are associated with each of the execution units. Note that in the case of Pentium II, the X86 based instruction set is translated into microoperations, which are fixed length instructions. Thus, the internal execution model of the processor can be significantly different from the external (user) view.

Instructions in the reservation can be executed "out of order" as long as they preserve the semantics of the program. Thus, each instruction can be either in "*ready*" state (i.e., all their inputs are "ready") or in "*wait*" state. Whenever an execution unit is ready to execute a new operation, it fetches a "ready" operation from its local

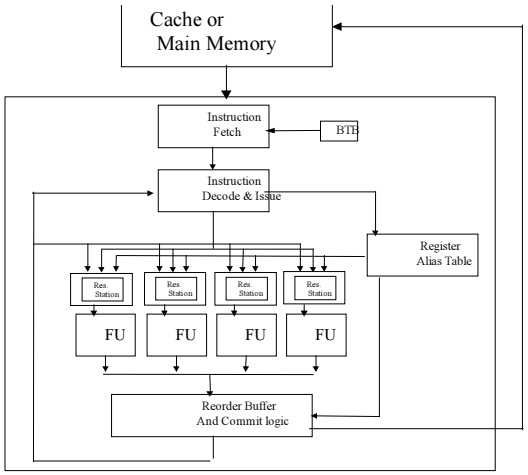


Figure 2.1 – Basic Architecture of Superscalar Out Of Order Processor

queue. No order is imposed between "*ready*" instructions.

When instructions are logged in the ROB, the Register Alias Table (RAT) determines if their source operands should be taken from the architectural register file or from the ROB. Each ROB entry keeps track of the program counter associated with its instruction and has a place to hold the execution result. The issue logic monitors the buses to detect writes to the ROB entries matching source operands of the waiting instructions.

The last pipeline stage in an out-of-order processor is instruction commit (retirement). Most current superscalar architectures ensure that the instructions modify the system resources, such as architectural registers (as opposed to renaming registers), caches, etc. in the same order as appears in the program or as generated by the compiler. Inside the micro-architecture, the instructions can be executed out of order, though the instructions need complete their executions in the program order.

In order to enhance performance, modern computer architectures use speculative execution which is based on different predictors (such as branch prediction). When the predictor fails, the retirement mechanism is also used for rolling back from exception or executing the wrong path. In such cases, the system needs to complete the execution of all the instructions prior to the roll-back point and to discard all results after that point.

3 Prior Approaches in Temporal Redundancy

The main goal of this paper is to develop low-cost, high performance support to timed redundancy techniques. In order to objectively evaluate the proposed techniques, we first discuss two existing architectural techniques for timed redundancy. For this discussion, we restrict this to pipelined, multi-threaded and superscalar

models which are currently the most commonly used architectural paradigms.

3.1 The Instruction Duplication Technique

A simple technique to increase the fault toletance coverage at run time was proposed in [9]. Here an instruction sent from the instruction decoder to the instruction window allocates two entries in the ROB table, and is consequently sent out twice for execution. At commit time, the results of the two operations are compared and "retired" from the system only if both operation obtained the same result, i.e., fault-free situations.

3.2 The AR-SMT Technique

An alternate approach to time redundancy was recently proposed in [5] and termed as AR-SMT (Active-stream/Redundant-stream Simultaneous Multi Threading). This architecture extends existing SMT [10,11] machines that can execute two threads in parallel. In SMT, instructions are fetched from two independent streams of instructions (threads) and these share the system resources. In traditional SMT machines, the results are written back to their ROB, and each thread retires its instructions independently.

The AR-SMT suggests executing each program twice, by using two threads; one that is considered as the primary and the other as secondary. These threads are not symmetric in their priority and in their execution-mechanisms. The primary thread runs as before but writes its results to a special hardware mechanism termed "*Delay Buffer*". This buffer is used to ensure that both threads run in a lock-step manner (with a delay of at most the size of the delay buffer), and that the results produced by both threads are identical.

In order to reduce the overhead incurred by the use of SMT architecture, the author suggests improvements over the traditional SMT architectures, such as: (1) In order to sustain instruction bandwidth, a Trace Cache for each thread is proposed. Trace Caches keep the information in decoded form and do not need to perform bus accesses as long as a hit is achieved in the Trace Cache, (2) Use of aggressive prediction mechanisms to predict branches [13] as well as values [14,15]. (3) Allows the second thread to use values produced by the first thread. Thus, the second thread can take advantage of the maximum parallelism available by the machine. The uniqueness of the AR-SMT approach is its observation that the first thread can be used as a perfect predictor to the secondary thread. Thus, the second thread can utilize the maximum parallelism on the machine and re-execute operations as soon as their inputs are in the Delay Buffer.

The AR-SMT paper indicates that by using all these optimization techniques, the overhead of using AR-SMT machine is relatively small compared to the use of other temporal redundancy techniques. Although [5] provides simulation results to indicate the AR-SMT performance enhancement over SMT etc., a closer look at the potential implementations of the AR-SMT machine reveals that it can suffer from several limitations that may result in significant overhead and loss in performance. Specifically, these include:

- The use of multi-threaded technology invariably suffers from reduced instruction bandwidth that limits its performance. Adding Trace Caches can ease the problem for those applications that can fit into the Trace Cache. But, Trace Caches are inherently large (since they keep the instructions in their decoded form), so on one hand increasing the size of the Trace Cache can result in an unacceptable increase in the die size; on the other hand, splitting the Trace Cache between the threads increases memory traffic in the machine and consequently limits its performance.
- An efficient use of the Delay Buffer as an accelerator for the second thread forces a feedback loop to the ROB (and the instruction scheduler). In machines that run at a very high clock rate (and utilize super-pipelines to achieve it), such feedback will cause a major delay between the two threads. It may cause a performance loss and force the use of large Delay Buffers as well.
- The use of the Delay Buffer suffers from a major performance loss whenever the system is required to rollback, such as in the case of branch mis-prediction, exception handling and transients.

These limitations can either increase the cost of the AR-SMT solution and/or restrict its performance significantly. On this review of existing temporal redundancy approaches, the current paper targets developing an alternate cost-effective, low-overhead temporal redundancy approach. We emphasize that reducing the performance overhead in fault-free scenarios has more real impact on system performance than the performance loss reduction in the presence of faults.

3.3 The Diva Approach

Recently, [20] presented a new concept in designing complex superscalar systems. The new architecture calls to replace the traditional retirement mechanism of out-of-order machines, with a "checker" that re-executes the instructions. This checker, on one hand, can be simple since it uses the data produced by the fast and complex machine, and on the other hand can run with a slower clock rate, since it can save the need to execute speculative operations.

A DIVA based processor could handle transients as

well, but it necessitates major architecture enhancements which are not cost/performance effective and beyond the scope of our discussion here.

4 The Out Of Order *Reliable* Superscalar (O3RS) Architecture

We now introduce facets of the proposed O3RS architecture that aims to achieve maximal fault tolerance coverage at a minimal cost to either the die area or the fault-free performance of the machine. The basic approach utilizes time redundancy to enable recovery from transient/soft errors. In order to minimize the extra cost in area and performance, we suggest enhancing an existing superscalar mechanism only if it cannot guarantee fault-free execution by using traditional techniques such as adding ECC circuits.

4.1 At What Level Should New FT Techniques be Integrated?

It is clear that in order to guarantee the overall correctness of the system the entire system should be protected by both error detection and recovery (or error correction) mechanisms. However, we argue that it does not mean that the same mechanism should be used throughout in order to protect the system. Since most advanced memory systems integrate Error Correction Codes (ECC) techniques as part of the structure of the memory, registers and internal buses, we would like to integrate these techniques as part of our solution. Thus, unlike other papers such as [8,9] that execute the entire program twice (temporal redundancy), we argue that it is sufficient to enhance the protection mechanism for those parts of the system that cannot be protected by existing/alternate techniques.

In this section we examine each stage of the execution pipe to determine if additional fault tolerance mechanisms are warranted for adding to them:

Fetching: The fetching mechanisms chiefly use memory and bus technologies. Thus, no extra mechanisms are needed to guarantee the correctness of the data and instruction that are fetched.

Instruction decoder: If the instruction decoder works as a table look-up (as in many RISC architectures), we do not foresee the need to add any redundancy. But, if the instruction decoder is more complex (like many of the X86 architectures) we consider duplicating the control logic.

RAT and ROB: Since the ROB is a cyclic buffer; we assume it to be protected using conventional mechanisms such as ECC. In order to protect the RAT we may need to duplicate its control (sharing the same data).

In order to add our new mechanisms, we will have to modify the structure of the ROB (this will be discussed later in Sections 5 & 6)

Execution units: We do not assume any modification of the execution mechanism (one may consider to add more execution units as explained later in Sections 5 & 6)

Retirement mechanism: We need to design a new retirement mechanism and to incorporate the "voting mechanism" between the pair of operations in order to guarantee their proper execution.

Write-back: As long as the write-back is controlled by the ECC, we do not need to modify it.

This stage-by-stage pipeline analysis indicates two issues, namely:

- We can maintain a superscalar (transient) fault-free system without the use of expensive multi-threaded technology.
- From the complexity and performance point of view, the efficient design of the execution pipeline is a primary design driver given that this aspect cannot be efficiently addressed by existing time-redundancy based approaches.

5 A Basic Description of the Error Recovery Mechanism

In this section we start introducing the proposed architectural mechanisms. These mechanisms are based on temporal redundancy; i.e., execute each instruction twice, and incorporate specific recovery mechanisms in case of branch mis-prediction or in the case of transients. The primary design driver is reducing overheads.

In order to achieve this goal we propose the following modifications to the "traditional superscalar structures presented in Sec. 3. Here we outline the O3RS modifications and detail them further in Sec. 6. The basic O3RS modifications include:

- Modify the ROB table so that each operation, which is not LOAD or STORE, will be executed twice. In order to achieve this, we propose to add status bit to each ROB entry that indicates if the instruction is being executed for its first time or for a second time. Note that we do not care if an instruction will be executed twice on the same functional unit or on a different functional units since we assume transient/soft errors; i.e., the probability of an error to appear in a functional unit does not depend on its history of errors.
- Although we execute each instruction twice, we do not require two entries in the ROB and we do not impose any implicit order between the executions of the two instances.
- The results of the two independent executions of the

same operation will be retired iff: (a) The two executions agree on the results. (b) All previous results were committed.

If an operation cannot be committed, we need to distinguish between two cases: (a) both computations produced different results, so we need to re-execute the instruction until its result will be verified. (b) the execution was speculative, and the speculation was found to be false. In this case, the system should be rolled back to the last committed point.

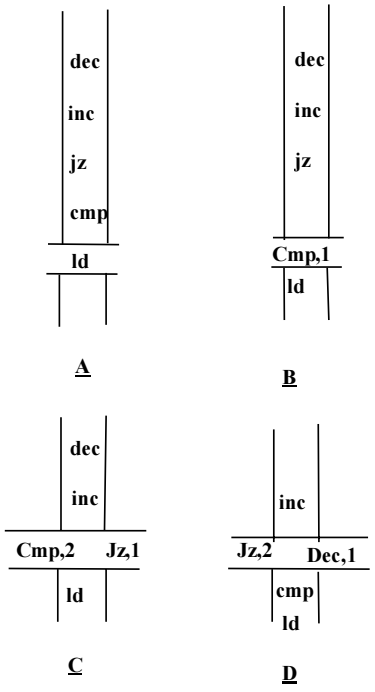
If an execution unit is repeatedly found to encounter errors, the system can take it out of the active list (if enough execution units are available)

In order to describe the operations in the proposed method, we present the following example:

```
a.  ld  R0,x          // load memory to R0
b.  cmp R0,$10        // compare R0 with 10
c.  jz   l1           // jump if zero
d.  inc  R0           // if R0 != 10 increment it
e.  l1: dec  R0       // else decrement it
```

As these instructions depend on each other, they must be executed sequentially even on an out of order execution machine. Let us assume that x contains 10, so the execution pipe of our machine will look as in Fig. 5.

Fig. 5 presents four steps of the execution pipe of the new architectural technique (a simplification of its operation). Each column in the figure represents the "stream of operations" (as is fetched into the instruction pipe), specifically the operations at the execution stage and the retirement stream. In this example, we assume no faults to occur.



Figures 5A-D: Basic O3RS Operations

Figure 5-A represents the start of the execution step; i.e., the first instruction (LD) is brought into the execution stage. Since load operation (LD) does not need to be verified via time redundancy mechanisms (as was explained in the previous section) we can move it immediately to the retirement stage (Figure 5-B), assuming a single cycle for load. At that point, the first instance of the compare instruction is forwarded to the execution phase.

Just after the first instance of the compare instruction is executed, the machine can now start its "second" execution in parallel with the first execution of the JZ operation (Figure 5-C). The first execution of the JZ indicates if the branch should be taken or not (in pipeline machine the branch prediction predicts the right direction before the JZ complete its operation). Since the first execution of the CMP indicates that the branch will be taken, it forwards the DEC instruction to the execution phase rather then the INC one. At that point (Figure 5-D), the JZ is being executed for the second time, and the DEC operation is being executed speculatively.

Note that during the second time an operation is executed, all of its inputs are known, so that it can start its execution immediately (assuming we have resource for that). If the system finds that the right answer to the compare instruction should be not-taken rather than taken, the system should roll-back (undo the DEC) and re-start the execution along the right path.

6 Hardware Enhancements For O3RS

In this section we provide a detailed description of the new hardware modifications needed to accomplish the proposed recovery mechanism. The goal of our proposed enhancements is to implement the new recovery mechanisms so it will result in minimal extra hardware that provides minimal overhead when no fault occurs. We will also show that the proposed recovery mechanism imposes low recovery overhead when the system needs to recover from either a transient/soft-error or from a branch mis-prediction, exception handling etc.

The implementation of the new algorithms utilizes the same basic superscalar structure used in previous sections. The instructions are fetched in a sequential order from the memory and are decoded in parallel. Each operation gets an entry in the ROB structure. As we have mentioned earlier, the ROB status bits are critical data and are protected extensively by ECC checks. The entries in the ROB are ordered in respect to their control dependencies. Each entry in the ROB contains in part information on the type of operation, what are the inputs (and if they are available) and a space to keep the

outcome of the operation. Each ROB entry contains also status bits that usually indicate:

E (Empty): The ROB entry is not attached to any operation.

W (Wait): The instruction is in the reservation station, and is waiting for inputs.

R (READY): The instruction is ready to be executed. (Moved to the reservation station)

D (Done): The operation is ready to be committed.

In order to incorporate the fault tolerant mechanisms, we suggest the following enhancements:

R: Indicates that the instruction is ready to be executed for the first time.

S: The instruction is ready to be executed for the second time.

D: The result of the two instances of the operation has been confirmed, and the operation is ready to be committed.

Note: (1) We ignore these extra bits if the instruction is LOAD or STORE. (2) Instructions do not need to check their dependencies when in S state. We could require that the second execution will start only if its sources are in D state. This might simplify the recovery mechanism but consequently reduce the overall performance due to the dependency resolutions. In the AR-SMT mechanism, it was suggested to use the delay buffer as a predictor for execution of the second thread. This mechanism complicates the hardware and may limit the frequency due to internal feedback loops. The O3RS mechanism achieves the same goal more efficiently and with simpler hardware.

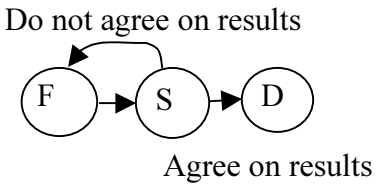


Figure 6.1 State transition of the ROB status

When the two executions of the same instruction do not agree upon the result, we suggest re-executing the instruction. A simple mechanism (as illustrated in the above state diagram: Fig. 6.1) calls to re-execute the instruction twice. A more sophisticated mechanism can either keep 2 output buffers and compare the third execution with both, or to copy the result of the second execution to the result buffer (in the ROB entry) and issue the instruction again in S state. As long as the error rate remains relatively low, we do not recommend complicating the hardware and believe that these simple mechanisms suffice. As mentioned earlier, it is more important to facilitate low performance overhead from the incorporated fault-tolerance mechanisms during *fault-free* operations than the overhead issues while handling errors.

When the system needs to change its control flow (as a result of error, mis-prediction, etc.) the operation of the system is very similar to the regular operation of any superscalar machine. The system needs to flush all the entries in the ROB that are below the mis-prediction point, regardless if they are being executed for the first time or for the second time. The system also needs to complete the execution of any instruction that is not in the D state and located above the mis-speculated point.

The retirement mechanism in O3RS remains the same as before; i.e., it "retire sequence" of instructions as long as all of them are in D state.

7 Performance Issues

At this stage we have described the basic functionality of the proposed O3RS approach. Prior to addressing the achievable performance issues, we again emphasize that although the transient rates, even at 0.1 micron process, are expected to be in order of a fault/day two points are worth noting, namely (a) it is unacceptable not to protect the machines against such faults (b) for systems that contain several processors the probability for error increases significantly. Thus, from the performance perspective, it is more important to make sure that the fault detection mechanisms do not cause a performance lost, than to optimize the performance of the recovery mechanism. With this background, the set of performance figures in this section will be focused on demonstrating fault-free overhead cases.

In order to evaluate the new system, we use a modified version of the SimpleScalar simulator [17] developed at the University of Wisconsin, and run our proposed techniques on different applications, using different set of inputs for each. The main performance measurement we were looking for was the utilization of the execution units. A utilization of 0.5 means that on average, all the resources were utilized half of the time. A utilization greater than 1 indicates that the new approach

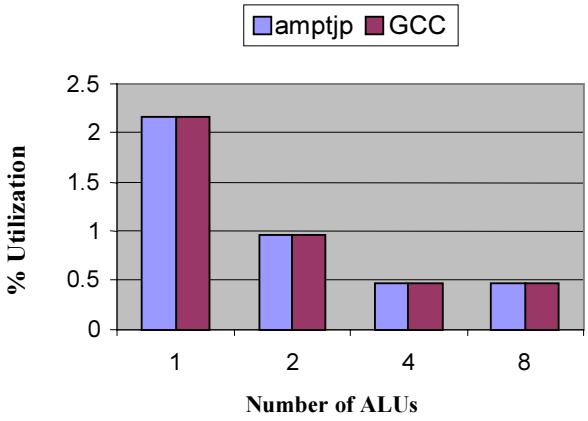


Figure 7.1 Performance of O3RS

slows down the system. Note that when a "regular" program is executed, the utilization corresponds to the CPI (Cycles Per Instruction) divided by the number of ALU's in the system. If the utilization is greater than 1, it means that the system was slowed down because of internal dependencies or due to system limitations.

Figure 7.1 presents the ALU utilization of the O3RS system running GCC compiler with 2 different inputs: GCC compiler (the compiler compiles itself) and AMPTJP program (both from INT SPEC 95). The Y-axis indicates the processor utilization and the X-axis (top row) indicates the number of ALU units. Each test runs for 100,000,000 simulated instructions. We can observe that (a) both inputs provide similar utilization numbers (b) if 2 ALUs (or more) are provided, the utilization is less then 1, meaning that there is no performance loss when applying the new technique. One can explain these numbers with the fact that the "second executed" instructions do not have to wait for inputs and so can be executed promptly, taking advantage of the relatively low ILP (Instruction Level Parallelism) of the GCC program.

Figure 7.2 presents the same utilization numbers for IJPEG program using 2 different inputs (Penguin and Vigo) taken from the SPEC 95 as well. The results presented here are similar to the result presented for the GCC run. The main difference is that in the IJPEG case, the system can well utilize up to 2 ALUs and only when four ALUs or more are used, that the extra cost for the new technique is diminished.

So far we mainly discussed the overall performance of the O3RS system and showed that it can take advantage of the current limitation in the available parallelism to execute the extra operations almost free of charge. Now we extend the discussion to other performance-related parameters of the system, such as instruction window, and fetch bandwidth.

From a performance viewpoint, one of the noticeable

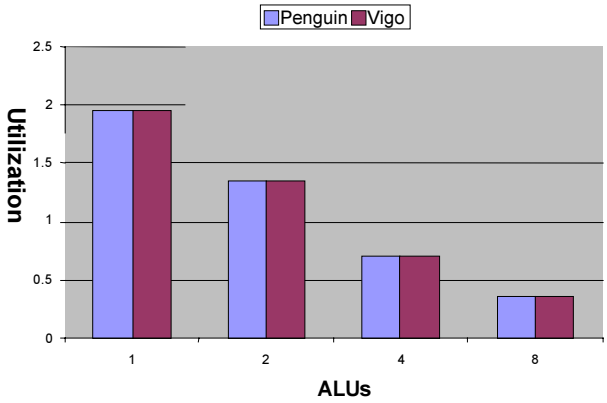


Figure 7.2 Resource Utilization Running IJPEG

disadvantages of the AR-SMT solution is that the implementation cost of an SMT machine is much higher, assuming the same level of performance. For example, in [5] it was assumed that each PE could fetch and decode up to 4 instructions each cycle. An implementation of such an instruction bandwidth is very difficult to achieve. On the other hand, limiting the instruction bandwidth can dramatically limit the overall performance of the system.

The [5,9] papers assume that each of the PE has its own trace cache and the number of the entries in the ROB for each thread remains the same regardless of the number of ALU's in the system. A more realistic assumption should be that the instruction window is split between the two threads in the system. Figure 7.3 shows the impact of the instruction window size (number of pending instructions) on the achievable ILP in the system. The X-axis measures the window size (8,16,32,64,128) for pending instructions, Y-axis measures ILP. The rows marked "MP3D" [solid line, bootm], "Walter" [dotted

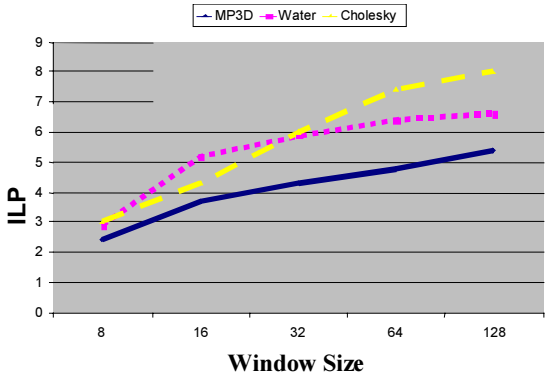


Figure 7.3: Impact of Instruction Window Size

line, middle] and "Cholesky" [dashed line] numerically represent the ILP values for these different applications.

Figure 7.3 demonstrates that by reducing the instruction window size from 64 ROB for a program to 32 entries for each thread, the system will lose about 15% of its performance (on average) for any SMT based approach. On the other hand, our proposed O3RS technique uses the same number of ROB entries as in "regular" superscalar. Consequently, it does not suffer from that performance degradation

The last chart we present (Figure 7.4) shows the slowdown of a single thread (out of 2 threads), assuming that the instruction window is split among them. In this experiment we assume 8% branch mis-prediction, and 5 cycles penalty for a branch miss. We also assume an 8 instruction fetch bandwidth.

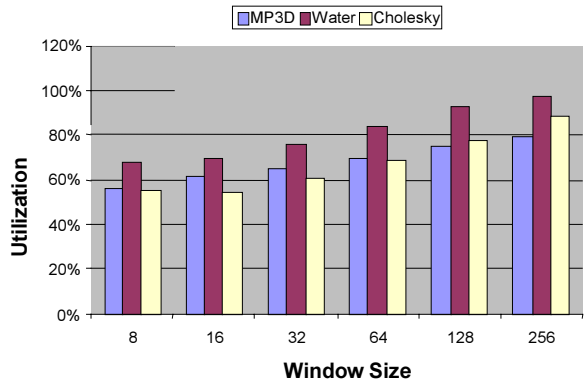


Figure 7.4: Utilization of a Single Thread

The chart indicates that on one hand, when the window size is relative small, the application is mainly limited by the instruction window size available for each thread. On the other hand, when the instruction window is large enough, the performance is limited by the instruction bandwidth and so the size of the window is less important.

8 Discussion & Conclusion

Overall, we have examined two basic time redundancy approaches of duplication and voting, and the associated integration of the fault-tolerance mechanisms within the structure of the superscalar architecture. An enhanced version of the recovery procedure that alleviates any performance loss over error recovery is also detailed.

Duplication of systems is the simplest implementation that may provide for good performance assuming that we can synchronize the duplicated instructions to occur in a lock-step manner. The drawback of this approach was identified to be the external addition of recovery mechanism, and the significant latency overhead. The duplication solution may also be costly in terms of hardware as we need to duplicate the entire system and the additional cost of voting and recovery.

The O3RS system approach presents a different outlook by integrating the fault tolerance capabilities as part of the internal structure of the superscalar architecture. The advantage of this approach is in enabling fast recovery mechanisms. The drawback is in flushing the execution pipe on error occurrences.

Looking at the ILP numbers, we can observe that computer architectures, such as the Pentium-II, generally have an average ILP of 1. Since such a machine is entitled to execute 2-3 ALU and address calculations per cycle, the average utilization of these units is less than half. Our proposal takes advantage of that and allows the second execution to use these free resources. Thus, in most cases the new O3RS mechanism will cause only minor performance slow down, if any at all.

When comparing the new O3RS system with similar techniques such as AR-SMT, the proposed O3RS technique presents a much simpler architectural and operational design that minimally impacts both the design effort and the performance. For example, the O3RS architecture does not use any Delay Buffers as was essentially required in the AR-SMT approach. The Delay Buffer can slow down the system in different ways one of which involves flushing the instructions from ROB and delay buffer. If such a flush operation is needed only for recovering from faults, we could use it as long as the probability for faults is low enough. Unfortunately, such operations are needed whenever branch a mis-prediction occurs and when the system experiences exceptions such as interrupts. Such operations occur often enough that they cannot be ignored.

Last, but not least, in this paper we assumed that transient errors are relatively infrequent thus leading to the application of basic time redundancy. However, if the soft-error rate increases, we will be able to incorporate more sophisticated recovery mechanisms such as the use of Roll-Forward recovery blocks [4] within O3RS. The roll-forward mechanism provides for sustained execution of dependent instructions as long as the inconsistency across replicates is not determined. When an error occurs, we save both replicate values and re-execute the operation again. If the new result commit with the first execution, the error was in the second execution, thus we can continue our normal execution assuming the original value was correct. If the new execution does not agree with the first execution, we need to roll back since there may be other operations in the execution pipe that based their uncommitted operation on a wrong result. We defer the details of the roll-forward blocks to a later document.

8.1 Conclusion

In this paper we have presented a novel approach for integrating fault tolerance capabilities within the structure of future microprocessor architectures. The basic driver is in providing for simple, low-cost mechanisms that provide coverage to transient faults which are likely to increase in occurrence as the device geometry shrinks and the level of device integration increases. Focussing on temporal redundancy techniques, our basic approach has been to look at various design alternatives in superscalar architectures such that the error recovery can be provided with minimal, if any, performance, design or operational overheads, and also with minimal modifications to the basic superscalar operations.

9 References

1. Diefendoff, K., "The Race to Point One Eight", *Microprocessor Report*, Vol. 12, # 12, pp. 10-22, Sept, 1998.
2. Johnson, M., "Superscalar Microprocessor Design", Prentice Hall, 1990.
3. Pentium II: Intel Architecture Manual, Intel Corp., 1997.
4. Mendelson, A. and Suri, N., "Cache Based Fault Recovery in Distributed Systems," *Proc. ICECCS*, pp. 19-129, 1997.
5. Rotenberg, E., "AR-SMT: A Microarchitecture Approach to Fault Tolerance in Microprocessors", Proc. FTCS-29, pp. 84-91, 1999.
6. Rubinfeld, P., "Virtual Roundtable on the Challenges and Trends in Processor Design: Managing Problems at High Speeds", *IEEE Computer*, 3(1):47-48, Jan 1998
7. Pradhan, D., "Fault Tolerant Computer System", *Prentice Hall*, 1998.
8. IBM 390x2 IBM System Journal, IBM Corp. 1996.
9. Sohi, G., Franklin, M. and Saluja, K., "A Study of Time-Redundant Fault Tolerant Techniques in High Performance Pipelined Computers", *Proc. FTCS-19*, pp. 436-443, 1989.
10. Tullsen, D., Eggers, S. and Levy, H. "Simultaneous Multithreading: Maximizing Chip Parallelism", *Proc. 22nd Intl. Symp. On Computer Architecture*, pp 392-403, 1995.
11. Tullsen, D., Eggers, S., Emer, J., Levy, H., Lo, J. and Stamm, R., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor" *Proc. 23rd Intl. Symp. on Computer Architecture*, pp 191 – 202, 1996.
12. Rotenberg, E., Jacobson, Q., Sazeides, Y. and Smith, J., "Trace Processors", *Proc. 30th Intl. Symp. on Microarchitecture*, Dec 1997.
13. Yeh, T. and Patt, Y., "Alternative Implementations of Two-Level Adaptive Branch Prediction", *Proc. 19th Intl. Symp. on Computer Architecture*, pp. 124-134, 1992.
14. Gabbay, F. and Mendelson, A., "Characterization of Value Prediction and its Impact on Modern Computer Architectures", *ACM Transaction on Computer Systems,* Vol 16, No 3, Sept 1998.
15. Lipasti, M., "Value Locality and Speculative Execution", PhD Thesis, Carnegie Mellon University, April 1997.
16. Anglada, R. and Rubio, A., "An Approach to Crosstalk Effect Analysis and Avoidance Techniques in Digital CMOS VLSI circuits", *International Journal of Electronics*, 6(5):9–17, 1988.
17. Burger, D. and Austin, T., "The Simplescalar Toolset, Version 2.0", Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.
18. Spainhower, L. and Gregg, T., "G4: A Fault-Tolerant CMOS Mainframe", Proc. *FTCS-28*, pp. 432-440, 1998.
19. Tamir, Y. and Tremblay, M., "High-Performance Fault Tolerant VLSI Systems Using Micro Rollback", *IEEE Transactions on Computers*, 39(4): pp. 548-553, April 1990
20. Austin, T., "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design", *Proc. MICRO-32*, pp.196-207,1999