

Error Detection and Recovery in High Performance Microprocessors prone to High Transient Error Rates

Aneesh Aggarwal

Department of Electrical and Computer Engineering

Binghamton University

Binghamton NY 13902

Email: aneesh@binghamton.edu

Abstract

With reducing feature size, increasing chip capacity, and increasing clock speed, microprocessors are becoming increasingly susceptible to transient (soft) errors. In this paper, we perform a detailed analysis of the various types of transient errors that can occur in a processor. We also propose various techniques (with different levels of aggressiveness) for transient error detection and recovery. The most aggressive technique maintains error checking codes for any data storage structure in a processor. These codes are used to detect errors when data is read from the structure, thus preventing the error from being propagated down the execution stream, before it is detected. In this technique, when an error is detected, it is also aggressively corrected. We show that as the error rate increases, aggressive error detection and recovery techniques become extremely important to maintain high performance.

In this paper, we also propose *efficient register renaming* and *branch reuse* techniques to improve the performance of processors with error detection and recovery capability. *Efficient register renaming* allocates consecutive registers to the multiple copies of the same instruction, and *branch reuse* technique reuses the branch outcomes of previous executions, during the recovery mode.

Keywords: High Performance Processors, Concurrent Error Detection and Recovery, Mean Time to Failure, Mean Instruction Error Rate

1 Introduction

With the current trends in transistor size, voltage and clock frequency, microprocessors are becoming increasingly susceptible to hardware failures. Hardware errors in the current technology are predominantly transient errors [3, 13] that occur randomly due to various reasons such as electromagnetic influences, alpha particle radiations, power supply fluctuations due to ground bounce, crosstalk or glitches, and partially defective components and loose connections. Transient errors do not permanently damage an electronic device, but cause a one-time change that vanishes when a new data (voltage) is applied. With the widespread

use of computer systems for applications that are critical to our health, safety, and financial security, ensuring reliable computing in commercial computer systems has become extremely important.

Transient hardware errors are troublesome because they elude most of the current testing methods. In addition, current trends suggest that transient errors will be an increasing burden for microprocessor designers [19], especially for target applications where the transient error rate can be unexpectedly high and a failure due to a transient error is unacceptable (such as space system and military applications). Hence, it is imperative to design run-time mechanisms to detect the transient errors and recover from them. A popular approach to detect and possibly correct transient errors is to use redundant computations [9, 4, 3, 14, 1, 8, 10, 11, 17, 18, 5]. In these techniques, the same application is run multiple times in a lock-stepped manner and the errors are detected by corroborating the redundant results, and are corrected by either using a majority policy (for systems with a redundancy of three or higher) or restarting the application (for systems with duplex redundancy).

An attractive approach is to use the same “single-threaded” processor (STP) – that executes a single thread at a time – to perform the redundant computations simultaneously [9]. In the STP approach, the redundant threads can either be executed in a staggered manner – where instructions of one thread slack those of the other – or in conjunction – where same instructions of the redundant threads are executed simultaneously. In machine configurations that are resource constrained, staggered execution of redundant threads can place significant pressure on the resources, resulting in performance degradation. In this paper, we focus on the configuration where the same instructions of the two threads are executed simultaneously.

This paper has two major contributions:

- First, this paper presents a detailed analysis of the various types of transient errors that can oc-

cur in a processor. Based on the analysis, it also proposes different error detection and recovery techniques (with varying levels of aggressiveness) that take into consideration the transient error type. This paper also shows that with increasing error rates, the more aggressive techniques significantly outperform the lesser aggressive ones.

- Error detection and recovery techniques can lead to significant performance loss in an STP processor because of the redundant computations required. In this paper, we propose *efficient register renaming* and *branch reuse* techniques to recover some of the performance loss. *Efficient register renaming* allocates consecutive registers to the multiple copies of the same instruction, thus reducing the pressure on the ROB and the commit width of the processor. *Branch reuse* technique reuses the branch outcomes of previous executions, during the recovery mode.

The rest of the paper is organized as follows. Section 2 discusses transient error categorization and the various error detection and recovery techniques. Section 3 presents the experimental results and analysis. Section 4 presents related work. Finally, in Section 5, we conclude.

2 Concurrent Error Detection and Recovery

In this discussion, we consider an STP configuration running one redundant thread for concurrent error detection. In this configuration, instructions are fetched and replicated to form an original thread and a replica thread. These threads are then executed concurrently. The results of the multiple copies of the same instruction, from the two threads, are compared when the instructions commit, and an error is detected if the comparison fails. In this discussion, we also assume that the caches and the memory are transient fault tolerant [13], and lie outside the sphere of replication [10].

2.1 Transient Error Categorization

The classification of transient errors is shown in Figure 1. We can broadly classify transient errors into two types: detectable errors (DE) and non-detectable errors (NDE). Some errors in PC generation and in the issue queue fall in the category of NDE errors. In an STP configuration, a PC generation error may never get detected because of a lack of redundancy in the fetch part of the processor. In this case, the program execution will continue in the wrong direction (WE type error). Similarly, an error in the issue queue can result in a deadlock (DL type error), and may never

get detected. For instance, if a register identifier of an instruction changes in the issue queue, the instruction may never get woken up. This will result in the program execution to stop and the error may not be detected.

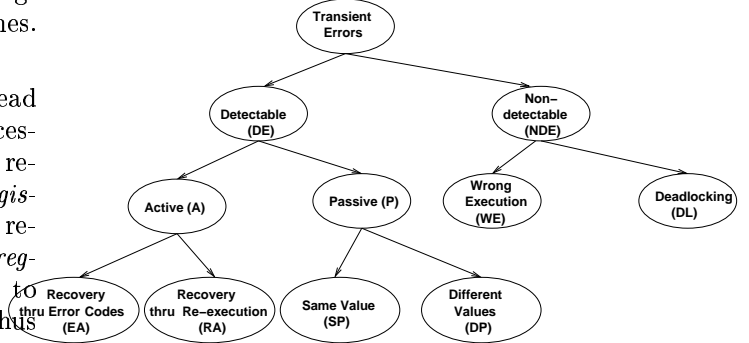


Figure 1: Transient Error Categorization

Detectable errors can be broadly classified into 2 types: active errors (A) and passive errors (P). Passive errors are errors that have occurred in the past, but have manifested during the current instruction’s execution. Such errors include errors occurring in data storage, such as change of data values in register files, rename map tables, branch target buffers, etc. For instance, consider that an instruction i_x writes to register R_x and commits without any errors. However, by the time a dependent instruction i_y reads R_x , the value in R_x changes. This will result in an error being detected in the execution of instruction i_y . A similar situation can arise if the entries in the map table change. It may not be possible to recover from passive errors by simply re-executing the faulty instructions, and reliable checkpointing is required. Active errors are errors that actually occur in the current instruction’s execution, and include control logic errors (such as decoder faults, MUX faults, register address faults, etc.), data computation errors (such as wrong result or effective address generation), data transfer errors (such as errors in data buses), and errors in some storage structures used to store data temporarily such as load/store queue, and issue queue. For instance, the opcodes in the issue queue and/or the addresses in the load/store queue can change, resulting in errors. Simple re-execution of faulty instructions can correct the active errors.

Passive transient errors can be further classified into 2 types: same value (SP) and different values (DP) passive errors. SP errors occur in data storages (*e. g.* register files) where the values written by the multiple copies of the same instruction are the same. DP errors occur in data storages (*e. g.* rename map tables) where the values written by the multiple copies of the same instruction are different. Active errors

can also be classified into 2 types: those that require re-execution (RA) for recovery (such as control logic errors, data computation errors, etc.) and those that can use error codes (EA) for recovery (such as load/store queue errors). EA errors occur when the multiple copies of the same instruction correctly write the same value into a data storage structure, but the values change while residing in the structure. However, EA errors occur in data storage structures where the values written by the instructions are not used by any other instructions. Table 1 gives the error categorization for some of the important hardware structures in a processor.

Hardware Structure	Transient Error
<i>Register Files</i>	SP
<i>Rename Map Table</i>	DP
<i>Issue Queue</i>	DL/RA
<i>Load/Store Queue</i>	EA
<i>Data Buses/ Control Logic</i>	RA
<i>Func. Units/ Latches/ Muxes</i>	RA
<i>ROB</i>	RA/DP
<i>Branch Predictor</i>	P

Table 1: Transient Error Type for Various Structures in an STP configuration

In Table 1, the branch predictor error is just a passive error because for an STP configuration, the branch predictor does not contain redundant information. Another data storage structure of interest is the ROB. For any instruction, the ROB stores both the current and the previous mappings for the destination register of the instruction. The *previous mapping* is used to recover from branch mispredictions and the *current mapping* is committed when the instruction commits. If an error occurs in the ROB, then two possibilities arise. If the instruction (in which the error occurred) is in the correct path, then during commit, the *current mapping* in the ROB can be incorrect, resulting in a wrong mapping being committed. This error can be corrected by re-executing the instruction and falls in the category of RA errors. However, if the instruction is on a misspeculated path, then rollback may be performed in an incorrect way because of an error in the *previous mapping*. This will result in passive errors in the following instructions that use the incorrect rename mapping. Since the *previous mappings* of the multiple copies of the same instruction are different, this error falls in the DP errors category.

2.2 Error Detection and Recovery

In this section, we discuss 3 error detection and recovery techniques in the ascending order of aggressiveness. First, we discuss the detection and recovery mechanism of deadlocking errors and active errors, which is similar in all the 3 techniques.

Detecting and Recovering Non-detectable Errors:

Special efforts are needed to detect the non-detectable errors. To detect a PC generation error, PCs are multiply generated and compared. To detect other deadlocking errors, such as the one in the issue queue, a *cycle counter* is maintained for each instruction in the ROB. When an instruction is allocated an entry in the ROB, the counter is *reset*. The counter is then incremented every cycle till the instruction commits or gets squashed. When the *cycle counter* reaches a predetermined threshold value, it triggers a recovery process, assuming that an DL error has occurred. DL errors are corrected by re-executing the instructions. The number of *cycle counters* can be reduced by sharing a counter among a set of ROB entries. In this case, the counter for a set of ROB entries is reset when the first ROB entry in the set is used. The counter then increments till there are valid instructions in the set.

Detecting and Recovering Active Errors: All active errors can be detected by comparing the results of the multiple copies of the same instruction, and can be corrected by re-executing the instructions. However, re-execution can be avoided for EA type active errors.

When the structures, where EA errors can occur, are written, error codes are generated for the values and stored. Subsequently, when the entries are read for both the original as well as the replica instructions (*e.g.* for comparing the addresses from the load/store queue or the register values from the additional register file), the values in the entries and their error codes are compared. If the values do not match and the error codes match, then the error codes are used to find the correct value. If the error codes also do not match, re-execution is initiated to recover from the errors. Note that, an error in a storage entry cannot be corrected by using just the error codes for that entry, even if the error codes have some error correcting capability. This is because, in case the error codes and the values do not match, it is not known whether the error occurred in the value or the code. Also, EA type active errors can be detected and corrected for only those storage structures where the values for the multiple copies of the same instruction are available simultaneously.

Following are the 3 error detection and recovery techniques:

- *Lazy Error Detection and Conservative Recovery (LDCR):* In this technique, error is detected only at commit time, by comparing the results of the redundant threads. When an error is detected, recovery is conservatively performed by reloading the checkpointed state and re-executing the instructions. This technique does not take into consideration the type of the error that occurred.

- *Lazy Error Detection and Aggressive Recovery (LDAR)*: In this scheme, error detection is performed as in the *LDCR* technique. When an error is detected, the instruction for which the error is detected and the following instructions are executed again to determine whether the error is an active or a passive error. In the subsequent executions, if no error is detected, then the error was an active error and the execution continues. If the error persists in subsequent executions, then the error is a passive error. In this case, the error is corrected by using the *checkpointed state*, and then the instructions are re-executed.
- *Eager Error Detection and Recovery (EDAR)*: *Lazy error detection* can result in significant performance loss (especially if the error rate increases) because many cycles can elapse between the time when an error occurs and when it is detected. In addition, the faulty instructions may have to be re-executed multiple times to determine the type of error. Hence, we propose *eager detection and recovery* of errors. It may be expensive to *eagerly detect* all the possible active errors such as errors in the control logic, Muxes, etc. Hence, we focus on *eagerly detecting* only the passive errors.

For *eager detection* of passive errors, when a value is written into a data storage element, an error code is also generated for the value and stored. We call this process *local checkpointing*. When the data is subsequently read, the error code for the data is compared against the value. If a value read from the data storage does not match with its error code, then a passive error is detected. The operation may have to be repeated to confirm the type of error (an active error from data transfer can result in a faulty passive error being detected). To *eagerly correct* a passive error in the register file, the values and codes for that particular register are read from the additional register file and compared. To *eagerly correct* a passive error in the rename map table, the *current rename mapping* is read from the last instruction which updated that particular rename map table entry. For this, the *ROB index table* stores the *ROB index* for the last instruction that updated a map table entry. However, the last instruction that updated a map table entry may have committed, in which case the *ROB index* will be invalid. To solve this issue, when an instruction commits, it sets a *committed bit* in the *ROB index table* entry if it is the last instruction that updated the entry. The *committed bit* enables correct recovery from a map table error. To restore the *ROB index*

table during branch misprediction rollback, *ROB* entries also record the *previous mapping ROB index*. In this case, *eagerly detected* passive *ROB* errors can be corrected using the *previous mapping ROB index*. If the *previous mapping ROB index* has committed, then the *committed bit* is set for the *ROB index table* entry. The *ROB index table* can be easily managed using some comparators.

If an *eagerly detected* error cannot be corrected (may be because all the entries in the different locations are corrupt), then the pipeline is accordingly stalled till either the instruction that detected a passive error becomes the oldest instruction in the *ROB* or is squashed. If the instruction becomes the oldest instruction, the error is then corrected using the *checkpointed state* and the instructions are re-executed. If no passive errors are detected and the instruction still errs at the commit stage, then it is an active error, which is corrected by re-executing the instructions.

Among the three techniques, *LDCR* is the least expensive technique in terms of hardware requirements and the most expensive one in terms of performance, whereas, *EDAR* is the most expensive technique in terms of hardware requirements and the least expensive one in terms of performance.

2.3 Fault Coverage in an STP

STP may not be able to cover faults in the fetch stage and the additional replication hardware. Note that, errors in the branch predictors used in the fetch stage will be detected as a branch misprediction, and corrected when the branch instruction updates the predictor. As discussed earlier, errors in PC generation are detected by replicating the PC generation hardware. To detect other errors in the fetch stage, and to detect and correct errors in the replication hardware, the error codes for the instructions stored in the instruction cache are read along with the instructions. Once the instructions are replicated, the multiple instructions and their error codes are compared. If the instructions match and the error codes do not match with the instructions, then an error occurred in reading the instructions and the instructions are re-fetched. If the instructions do not match, then an error was introduced by the replication hardware and can be corrected using the error codes. If a replication error cannot be corrected, the instructions are re-fetched from the cache.

2.4 Efficient Register Renaming

In an STP processor with a 2-way redundancy, the effective *ROB* size reduces to half of the physical *ROB* size. This can lead to a significant performance loss.

Note that, when using redundant computations, registers for the original and the replica instructions are allocated and deallocated simultaneously. This suggests that the two instructions can be allocated consecutive registers, without increasing the register pressure. In this case, the rename mapping for the replica instruction can be derived from that of the original instruction. We call this renaming technique *Efficient register renaming (ERR)*. *ERR* allows us to use a single ROB entry for both the original as well as the replica instructions. This reduces the pressure on the ROB, which can result in significant performance improvement.

ERR also benefits by avoiding the use of a separate rename map table for the replica instructions in the *EDAR* technique. Note that the errors in the rename map table in the *EDAR* technique are detected using error codes. *ERR* technique also reduces the size of the *ROB index table*. *LDCR* and *LDAR* will still require multiple rename map tables, even with *efficient register renaming*, to detect errors in the rename map tables.

ERR can be easily implemented by partitioning the register file into 2 parts. Registers are allocated to the original instructions from the first part and the corresponding registers are allocated to the replica from the second part. *ERR* also helps in reducing the number of register file read ports, because the original instructions read the values from one partition and the replicas from the other. Reduction in the register file ports can translate into clock cycle improvements. If the number of register read ports are not reduced in the original register file, then *ERR* can avoid the use of an additional register file. For this, each register is associated with a *status bit* that is reset when a register is deallocated and set when the register is written. At the time of commit, multiple copies of an instruction read their values from the corresponding register file parts, for checking the values and checkpointing them. If a passive register error is *eagerly detected*, then the instruction checks the *status bit* of the register written to by the other copy. The thread which detected a passive register error is stalled till the status bit of the register is set. When the status bit is set, the instruction uses the value in the register to recover from the error. However, stalling the thread can lead to deadlock where one thread stalls on one register and the other thread stalls on another register. This deadlock is detected by the *cycle counters*, and the instructions are re-executed. Another approach is to not stall the thread that detected the error, in which case the scheduled instructions dependent on the instruction that detected an error will need to be “unscheduled”. If the additional register file is still used and the register read ports are not reduced, then

a reduction in the effective commit width of the processor can be avoided, which can further reduce the performance impact due to redundant computations.

2.5 Branch Reuse for Fast Recovery

The recovery process for many of the transient errors requires re-executing the instructions. This may also result in the re-execution of some branch instructions. Note that the branches update the branch predictor at commit time. Hence, a branch that was mispredicted during the first execution will also be mispredicted during re-execution. However, the outcome of some of the re-executed branch instructions may be known from the earlier execution, and these branch outcomes can be used for better branch prediction during re-execution. Hence, we propose that the branch evaluations be stored in a small circular FIFO buffer called the *branch reuse buffer*. Each *branch reuse buffer* entry consists of a *valid bit* indicating whether the entry is valid or not, an *evaluated bit* which indicates whether the branch is evaluated or not, an *outcome bit* which gives the branch outcome and the *target address bits* which give the target address of the branch.

When the branches are dispatched, they are allocated the last entry in the buffer. Simultaneously, the *valid bit* is set and the *evaluated bit* is reset. When the branches commit, they are removed from the top of the buffer. On a branch misprediction, the branches younger to the one that mispredicted are squashed from the *branch reuse buffer*. When a branch is evaluated, its evaluation is stored in the *branch reuse buffer* only if it is mispredicted. Otherwise, nothing is done. To record the branch outcome, branch instructions also carry the index into the *branch reuse buffer*. When re-executing the branch instructions, the branch outcomes from the *branch reuse buffer* override the predictions from the branch predictor. If a misprediction is detected by the *branch reuse buffer*, then the following entries in the buffer are invalidated because the program execution now flows in a different direction. The size of the *branch reuse buffer* depends on the maximum number of outstanding branches that can be present in the processor. We use a small 16-entry *branch reuse buffer* in our experiments. Since the *branch reuse buffer* is small, it may get filled up. In such a case, if more branches are dispatched, incorrect *branch reuse* may result. To avoid incorrect *branch reuse*, we use a small counter to count the number of branches dispatched after the buffer is full. Accordingly, branch instructions record a bit indicating whether they have an entry in the *branch reuse buffer* or they form a part of the count in the counter. When a branch instruction is committed or squashed, depending on the bit, either an entry is deallocated from the buffer or the counter is decremented. Note

that errors occurring in the *branch reuse buffer* will be detected when the branch is evaluated, as is the case for branch predictor errors.

3 Reliable STP Processor

The processor architecture in this section is discussed for *LDAR* error detection and recovery. The additional operations to be performed for *EDAR* and *LDCR* are *italicized*. Figure 2 shows the schematic diagram for our processor. The instructions and their correcting codes are fetched from the instruction cache. While the instructions are fetched and the new PCs generated, multiple copies of the old PCs are checked for errors, and in case of an error, fetch is re-initiated. *For eager detection of errors in branch predictor, the predictions from the predictor are checked with the locally checkpointed error codes. If an error is detected, the fetch is stalled till the branch is evaluated.* When in recovery mode, the branch outcomes from the *branch reuse buffer* override the predictions from the branch predictor. The instructions are then replicated to form two independent threads of execution. The replicated instructions are then checked for instruction fetch and instruction replication errors. Both the original and the replica instructions are then decoded and renamed using *Efficient Register Renaming*. Pipeline is stalled if registers cannot be allocated. The physical registers used by the replicas are freed in the same manner as those used by the original instructions. *In EDAR technique, during renaming, the rename mapping read from the map table is checked with its error codes, and errors are corrected as discussed in Section 2.3. Error code is also generated for the new rename mapping updated into the map table.*

The instructions are then dispatched to their respective subsystems (integer instructions to integer subsystem and FP instructions to FP subsystem). Simultaneously, both the original and the replica instructions are also allocated a single entry in the ROB, and consecutive entries in the load-store queue for the load and store instructions. Error codes are also generated for the rename mappings stored in the ROB entries. The two threads (the original and the replica) execute independent of each other. *In EADR, the values read from the register files are also compared against the checking codes, and in case of an error, the error is corrected. When a register result is produced, error codes are also generated and stored.* The generated register results are also stored in the additional register file along with the error codes. The addresses generated are stored in the load/store queues along with the error codes for the addresses. The branch outcomes are also stored in the *branch reuse buffer*. When the original instruction and its replica are ready

to commit, their results are corroborated to detect any errors. If no errors are detected, the results of the instructions and their rename mappings are committed and checkpointed. If an error is detected, the execution is restarted from the faulty instruction to determine the type of error. If the error is repeated (indicating a passive error), the error is corrected using the checkpointed state, and the instructions are re-executed. *In EDAR, if an error is detected, then it is an active error and the execution of the instructions is restarted. In LDCR, if an error is detected, the checkpointed state is reloaded and the instructions are re-executed.*

In these techniques, the load and store instructions are handled in a slightly different manner. Since the caches and memory are transient error tolerant, only the original store instruction stores the value, and only the original load instruction loads the value and forwards it to the registers allocated to both the original and replica load instructions. However, the addresses and the values of the loads and the stores are checked for errors. Note that, load instructions are issued only when the addresses of the multiple copies of the load instructions are available.

4 Experimental Results

4.1 Experimental Setup

The hardware parameters for the base superscalar processor, without redundancy, are given in Table 2. Our base pipeline consists of 9 front-end stages. Two pipeline stages are inserted for instruction replication and error checking for reliable STP configuration. We use a modified SimpleScalar simulator [2], simulating a 32-bit PISA architecture. In our simulator, instead of having a unified RUU depicting the issue queue, register file and ROB, we have separate ROB, issue queues, and register files. We use a unified physical and architectural register file where the architectural registers are committed in the physical register file itself. For benchmarks, we use a collection of 5 SPEC2000 integer (*vpr*, *mcf*, *parser*, *bzip2*, and *gcc*), and 8 FP (*wupwise*, *applu*, *art*, *ammp*, *swim*, *equake*, *mgrid*, and *apsi*) benchmarks. The statistics are collected for 1B instructions after skipping the first 2B instructions.

Before measuring the performance of the various error detection and recovery techniques, we measure the performance of our STP configurations with *efficient register renaming* – without additional register file (*STPNARF*) and with increased commit width (*STPCW*), as discussed in Section 2.5. The experiments are performed without inducing any errors. We compare their IPC results against the base superscalar configuration (which does not perform any error de-

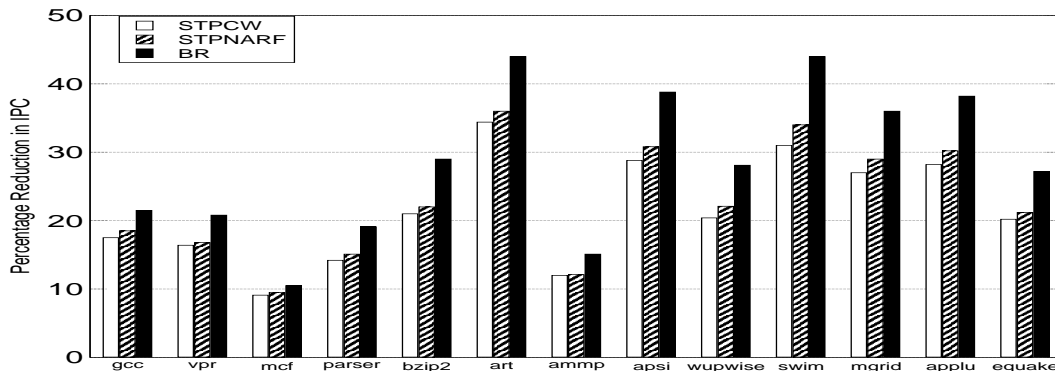


Figure 3: Performance Reduction in IPC of Spec2k Benchmarks

marks. This results in the reduction in effective issue queue size, load/store queue size, and dispatch width having a significantly more performance impact than they have in integer benchmarks. In addition, better branch prediction and higher register pressure in FP benchmarks exacerbates the performance impact. The second observation from Figure 3 is that the performance improvement with our STP configurations is more for the FP benchmarks. The reason for this is the higher pressure on the ROB and the commit width observed for the FP benchmarks. This pressure is relieved with our STP configurations.

4.3 Performance of LDCR, LDAR, and EDAR

In this section, we measure the performance of the *LDCR*, *LDAR*, and *EDAR* techniques in the presence of errors. In our experiments, we randomly generate errors in the various data storage elements and the pipeline stages. Errors are also generated in additional data storage elements used in some of the techniques such as the additional register files, buffers storing the error codes, branch reuse buffer, and the ROB index buffer. The maximum number of bit-errors that can be detected by the error codes used in our experiments for the various data storage structures are shown in Table 3. The error detection capability for the various structures will depend on the reliability expected out of the processor. The detection capability for the ROB errors is high because the current mapping read from the ROB is checkpointed and even large number of errors should be detected in the mapping. On the other hand, the detection capability for the branch predictor errors, and the branch reuse buffer errors is low because even if these errors are detected, not much can be done other than to stall the front-end pipeline till the errors can be corrected. Similarly, the codes used for the additional register files and the load/store queues can be small because the values are also compared in these buffers and any error can be detected. However, if the values do not match and the

error codes do, then the codes should be able to determine the correct value, which may require the codes to detect larger number of errors. Hence, for these buffers, we provide a code with larger error detection capability. The rename map table and the original register file errors can be eagerly detected and corrected as discussed in Section 2.3, and hence a larger error detection capability can avoid performance loss. Our experiments also showed that the error detection capability of Table 3 is more than enough for correct functioning of the processor, even with a very low mean time to failure (MTTF) for each structure.

Storage Structure	Max. Errors
<i>ROB</i>	3
<i>Branch Predictor</i>	1
<i>Original Register Files</i>	3
<i>Additional Register Files</i>	3
<i>Map Table</i>	3
<i>Issue Queue</i>	2
<i>Branch Reuse Buffer</i>	1
<i>ROB Index Table</i>	3

Table 3: Error Detection Capability of Error Codes for Various Storage Structures

Figure 4 gives the percentage reduction in IPC for the *LDCR*, *LDAR*, and *EDAR* techniques. We only present the results with the experimental model with MTTFs of 10000, 1000, and 100 cycles. Very low MTTFs are expected in future technologies [6] and in harsh environments. For these results, all the structures are assumed to have the same MTTF. For the *LDCR* technique, we assume that it requires 200 cycles to load the checkpointed state in case of an error. For the *LDAR* and the *EDAR* techniques, we assume that it requires 5 cycles to correct a passive error. The base case (with respect to which the percentage reduction in IPC is shown) in these experiments is the *STPCW* technique. Figure 4 shows that the *LDCR* technique has about 3%, 20%, and 65% reduction in IPC for

an MTTF of 10000, 1000, and 100 cycles respectively. The corresponding numbers for the *LDAR* technique are about 0%, 5%, and 35%, and for the *EDAR* technique are about 0%, 2%, and 15%. Figure 4 suggests that as the transient error rate increases, *eager detection and recovery* of the transient errors can be much more beneficial. We also experimented with different cycle requirements for passive error correction for the *LDAR* and *EDAR* techniques, and found that the relative performance of these two techniques is quite independent to the cycle requirements.

5 Related Work

Techniques that simultaneously execute multiple copies of the same stream of instructions have been proposed for concurrent error detection and recovery [9, 1, 8, 10, 11, 17, 18, 5]. Ray, Hoe, and Falsafi [9] use the same superscalar datapath to execute the multiple copies of an instruction for fault-tolerance. Austin proposes a very different fault-tolerant scheme [1] which comprises of an aggressive out-of-order superscalar processor checked by a simple in-order checker processor. The fault-tolerant architectures in [10, 11, 18, 5] use the inherent hardware redundancy in simultaneous multithreading and chip multiprocessors for concurrent error detection. Patel and Fung [8] propose transforming the input operands between redundant computations and comparing the results to expose a persistent fault. These techniques assume that transient errors can be corrected by simply re-executing the instructions. Instead, in this paper, we argue that there are some transient errors that require checkpointing for recovery, and that there may also be some transient errors that are non-recoverable. We propose techniques to recover from these errors. In addition, the techniques in this paper also exploit the different types of transient errors for error detection and recovery.

Christopher Weaver [19] also realized that need to transform the hardware error rate into instruction error rate, to measure the performance impact of failures in high performance processors. They use the top-down approach for this purpose, whereas we use a bottom-up approach. Their approach derives the instruction error rate by considering the total number of instructions and the error rate of the entire processor, and factoring it with the vulnerability of the architecture [7] to the error. Our approach derives the instruction error rate using the combined effect of the error rate of each hardware structure inside the processor. We also corroborate our analytical error model with an experimental error model.

The *branch reuse buffer* discussed in this paper is somewhat similar to the branch outcome buffer sug-

gested in [18]. [18] uses the branch outcome buffer to forward the branch outcomes between the redundant threads, where one thread is trailing the other thread. Instead, we use the *branch reuse buffer* to forward the branch outcomes during re-execution, and also enable the buffer to handle cases where the number of outstanding branches are significantly higher than the size of the buffer.

6 Conclusion

With the current trends in transistor size, voltage, and clock frequency, microprocessors are becoming increasingly susceptible to transient (soft) failures. Even though high-end servers provide some mechanisms to ensure system reliability, reliable commodity microprocessors are becoming a must with the dissemination of computers in everyday life. Reliability in systems is usually ensured by corroborating the results of redundant computations.

In this paper, we first perform a detailed analysis of the different kinds of errors that can occur in a processor. Then, we propose different error detection and recovery techniques (with different levels of aggressiveness) that take into consideration the type of transient errors while recovering from the error. The most aggressive technique eagerly detects transient passive errors, using *local checkpointing*, and aggressively corrects them. Our studies showed that as the transient error rate increases, it may be much more beneficial to go for a more aggressive technique for error detection and recovery. Among the different error detection and recovery techniques discussed in this paper, the most aggressive technique outperforms the least aggressive one by about 50%. Redundant computations can lead to a significant performance loss in a “single threaded” processor architecture. In this paper, we also discuss *branch reuse* and *efficient register renaming* techniques to recover some of the performance loss. *Branch reuse* technique reuses the branch outcomes of earlier executions, in the recovery mode. *Efficient register renaming* performs renaming in such a manner that the degrading effects of redundant computation on the ROB size and the commit width are avoided. *Efficient register renaming* reduces the performance impact of redundant computations by about 10%.

References

- [1] T. Austin, “DIVA: a reliable substrate for deep submicron microarchitecture design,” *Proc. Micro-32*, 1999.
- [2] D. Burger and T. M. Austin, “The SimpleScalar Tool Set, Version 2.0,” *Computer Arch. News*, June 1997.

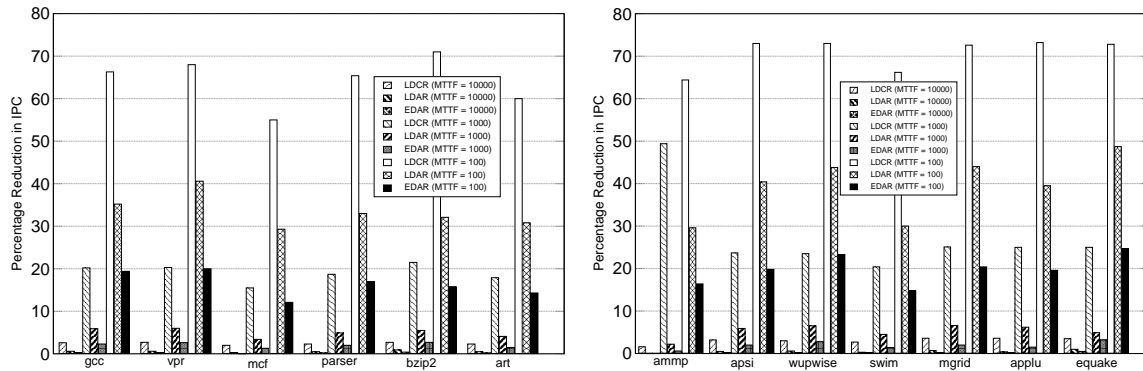


Figure 4: Performance of Spec2K (a) Integer and (b) FP benchmarks with various Error Detection and Recovery Schemes with MTTF = 10000, 1000, and 100 cycles

- [3] Compaq Computer Corp., “Data integrity for Compaq Non-Stop Himalaya servers,” <http://nonstop.compaq.com>, 1999.
- [4] J. G. Holm, and P. Banerjee, “Low cost concurrent error detection in a VLIW architecture using replicated instructions” *Proc. ICPP-21*, August 1992.
- [5] M. Gomma, et. al., “Transient-Fault Recovery for Chip Multiprocessors,” *Proc. ISCA-30*, 2003.
- [6] T/ Karnik, et al, “Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18um,” *IEEE Symp. VLSI Circuits*, 2001.
- [7] S. Mukherjee, et. al., “A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor,” *Proc. Micro-36*, 2003.
- [8] J. H. Patel, and L. T. Fung, “Concurrent error detection in ALU’s by recomputing with shifted operands,” *IEEE Transactions on Computers*, 31(7):589-595, July 1982.
- [9] J. Ray, J. Hoe, and B. Falsafi, “Dual use of superscalar datapath for transient-fault detection and recovery,” *Proc. Micro-34*, 2001.
- [10] S. Reinhardt, and S. Mukherjee, “Transient fault detection via simultaneous multithreading,” *Proc. ISCA-27*, June 2000.
- [11] E. Rotenberg, “AR-SMT: A microarchitectural approach to fault tolerance in microprocessors,” *Proc. of the 29th Intl. Symp. on Fault-Tolerant Computing Systems*, June 1999.
- [12] S. S. Sastry, S. Palacharla, and J. E. Smith, “Exploiting Idle Floating-Point Resources For Integer Execution,” *Proc. PLDI*, 1998.
- [13] D. P. Siewiorek and R. S. Swarz, “Reliable Computer Systems Design and Evaluation,” *The Digital Press*, 1992.
- [14] T. J. Slegel, et al. “IBM’s S/390 G5 microprocessor design,” *IEEE Micro*, 19(2):12-23, March/April 1999.
- [15] P. Shivakumar, et. al. “Modelling the effect of technology trends on the soft error rate of combinatorial logic,” *Proc. Dependable Systems and Networks (DSN)*, 2002.
- [16] M.L. Shooman, “Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design,” *Wiley Publications*, 2002.
- [17] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, “Slipstream processors: Improving both performance and fault tolerance,” *In Proc. Micro-33*, December 2000.
- [18] T. Vijaykumar, I. Pomeranz, and K. Cheng, “Transient-fault recovery using simultaneous multithreading,” *Proc. ISCA-29*, 2002.
- [19] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt, “Techniques to Reduce the Soft Error Rate of a High Performance Microprocessor,” *Proc. ISCA-31*, 2004.
- [20] The National Technology Roadmap for Semiconductors, Semiconductor Industry Association, 2001.