Instruction Error Rate Modeling for High Performance Microprocessors

Aneesh Aggarwal
ECE Department
Binghamton University
Binghamton, NY 13902
aneesh@binghamton.edu

Abstract

With reducing feature size, increasing chip capacity, and increasing clock speed, microprocessors are becoming increasingly susceptible to transient (soft) errors. Traditionally, soft error rates have been defined in terms of the rate at which the hardware generates erroneous results due to faults. However, this definition does not fit a high performance microprocessor model, consisting of various hardware modules and executing instructions. This is because a transient error in a hardware module does not necessarily generate a fault in program execution, because the faulty hardware may not be in use when an error occurs. In addition, the faulty instruction resulting from the faulty hardware may not contribute to the final result of the program. Hence, it is imperative to transform hardware error rates into instruction error rates that can be used in a high performance processor.

In this paper, we introduce a detailed analytical transient error model that transforms hardware error rates into instruction error rate. The model decomposes the processor into different functional entities each with its own *mean time between failures (MTBF)*, and derives a *mean instruction error rate (MIER)* metric that gives the error rate in terms of failed instructions per unit time. We compared the *MIER* generated by the analytical model with that generated by an experimental model and found that the analytical model gives an accurate measure of the instruction error rate. We then apply the analytical model to study how different hardware parameters affect the error generation in a high performance microprocessor. We also apply the model to compare the performance of various error detection and recovery techniques.

Keywords: High Performance Processors, Concurrent Error Detection and Recovery, Transient Error Models, Mean Time Between Failures, 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 [17], 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).

Transient hardware error rates are measured in terms of transient errors generated in the hardware per unit time. This transient error model does not fit the high performance microprocessor model. High performance microprocessors execute instructions using an enormous amount of hardware which can generate transient errors. However, every transient hardware error generated in a microprocessor will not necessarily result in a faulty execution, because the hardware that generated an error may not be in use at the time when the error was generated. For instance, a transient error that flips a bit in one of the inputs of an ALU that is not being used will not result in a faulty

execution. Even if a transient error results in a fault in an instruction, the faulty instruction may not contribute to the final outcome of the program, thus not resulting in a faulty execution. For instance, transient errors generating faults in instructions whose results are not used and in instructions along a mispredicted path. In a high performance microprocessor, transient faults do not matter as long as the final outcome of the program is correct. Hence, it is imperative that the hardware error rate is transformed into instruction error rate.

In this paper, we present a detailed analytical transient error model that transforms the hardware error rates into instruction error rate. This model derives a single *mean instruction error rate (MIER)* metric that gives the error rate in terms of failed instructions per unit time, by decomposing the processor into different functional entities each with its own *mean time between failures (MTBF)*. Such a metric is important because the rate at which instructions fail is required to measure the performance of any error detection and recovery technique used in high performance processors. For instance, consider that a new error detection and recovery technique is developed for register files that make the register files transient error tolerant. In such a case, the contribution of the register files to *MIER* (in the analytical model) can be discarded to get the new instruction error rate. In addition, *MIER* can be useful in determining the *MTBFs* of the various functional units for a target overall reliability of a high performance microprocessor. For instance, if a processor is to be used for a certain application, and once the behavior of the application and underlying susceptibility of hardware modules to failures is known, the processor can be designed accordingly to give better reliability.

We also check the correctness of the analytical model by inducing errors in a cycle-accurate high performance microprocessor simulator. In our experiments, we induce errors in each of the functional units depending on the *MTBF* of that unit and random number generation. We found that the *MIER* calculated using the analytical model follows very closely to the instruction error rate obtained for the experimental model. We then apply the analytical model to study how different hardware parameters affect the number of errors generated in a microprocessor, and to compare the performance of various error detection and recovery techniques.

The rest of the paper is organized as follows. Section 2 discusses the analytical and the experimental transient error models for a high performance microprocessor. Section 3 presents the results of experiments conduct to corroborate the analytical model. Section 4 discusses the performance of microprocessors in the presence of errors. Section 5 applies the analytical model to study the effects of various hardware parameters on error generation. Section 6 compares the performance of various error detection and recovery techniques. Section 7 presents related work. Finally, in Section 8, we conclude.

2 Transient Error Models

2.1 Analytical Model

In performing the reliability analysis of a processor, it is almost impossible to treat the processor in its entirety because of its complexity. Hence, the logical approach is to decompose the processor into different functional entities. For an instruction to commit without any errors, all the functional entities used by the instruction should be correct. This will include functional entities in both the datapath and the control path of the instruction. So, the probability that an instruction commits without any errors can be given by:

$$P = \prod_{i=1}^{n} P(x_i) = \prod_{i=1}^{n} e^{-\lambda_i t} \qquad (1)$$

$$\lambda_i = \frac{1}{MTBF_i}$$

where x_1 to x_n are the different hardware structures used by the instruction, $P(x_i)$ is the probability that the hardware structure x_i does not have any errors, $MTBF_i$ is the mean time between failures of structure x_i, and λ_i is the failure rate of structure x_i. λ_i depends on the radiation flux rate, the technology parameters, and the implementation of the circuit. Equation (1) assumes that the errors in the different hardware structures are independent of each other, and that each structure exhibits a constant failure rate. However, if all the hardware entities (such as latches and multiplexors) used in the execution of an instruction are considered separately, the processor may have to be decomposed into millions of entities. This will make the model very cumbersome.

In our error model, we partition the processor into different data storage structures (R) each of size S, where each structure has its own failure rate λ_R. λ_R only includes errors that change the stored value in a data storage structure (once the value has been written in the structure). So, the failure rate of each entry in a data storage structure R of size S will be $\frac{\lambda_R}{S}$, assuming an equal distribution of the failures among all the entries. The remaining active elements in the

processor (such as the control logic, the muxes, the wires, etc.), where a transient error can occur are combined together into a single structure. We call this structure the *active structure* (A), with a failure rate λ_A. Note that λ_A also includes failures in the process of writing into and reading from the data storage elements. There is an active structure for each pipeline stage of the processor, consisting of the active elements used in that pipeline stage. However, for a pipeline stage of width W (the pipeline stage can process W instructions in parallel), an instruction functions correctly if the pipeline used by the instruction is correct. Hence, for a single instruction, the active structure error rate for an active structure A of width W will be $\frac{\lambda_A}{W}$. Our model also assumes that transient errors in different hardware structures are independent of each other, and that each structure exhibits a constant failure rate. The probability that an instruction commits without any errors now becomes:

$$P = (\prod_{i=1}^{n} P(r_i))(\prod_{j=1}^{k} P(a_j)) \quad (2)$$

$$P = (\prod_{i=1}^{n} e^{\frac{-\lambda_{R_i}}{S_i}t})(\prod_{j=1}^{k} e^{\frac{-\lambda_{A_j}}{W_j}t}) \quad (3)$$

$$P = exp(-\sum_{i=1}^{n} \frac{\lambda_{R_i}}{S_i}t - \sum_{j=1}^{k} \frac{\lambda_{A_j}}{W_j}t) \quad (4)$$

where r_i depicts the unique data storage structure entries accessed by the instruction, and a_j depicts the pipelines used by the instruction. Assuming a constant instruction error rate (in terms of instructions failed per unit time), the instruction error rate can be given from equation (4) as $\sum_{i=1}^{n} \frac{\lambda_{R_i}}{S_i} + \sum_{j=1}^{k} \frac{\lambda_{A_j}}{W_j}$.

The instruction error rate considers the hardware errors that can occur at the instant the hardware is used by the instruction. However, once a storage entry is written, an error that occurs in the entry in any cycle till the entry is read by an instruction will fail the instruction. For instance, consider that a register is written at time t, and an error occurs in the register at time $t + x$. Even though, no instruction is reading the register at time $t + x$, any instruction that reads the register after time $t + x$ will fail. This suggests that the error rate of a storage entry should be multiplied by the average number of cycles the storage entry is in use. We denote this number of cycles by C. For instance, if a register is used by 2 instructions, where one instruction reads the value 2 cycles after it is written and the second instruction reads the value 4 cycles after it is written, then for this particular storage entry, $C = 3$. C is equal to *zero* for the cases when the data storage entry is never used once it is written. Note that, if the storage structure is a queue where instructions wait (such as an issue queue or the ROB or the load/store queue), the C signifies the number of cycles spent by an instruction in the queue.

Active structures comprise of two components: the static logic and the latches, and the error rate of an active structure (λ_{A_j}) is the addition of the error rate of the static logic and that of the latches. At present, the contribution of the static logic to the overall error rate is assumed to be small [7]. Hence, for all practical purposes, λ_{A_j} can be assumed to be the error rates in the latches (most of which are utilized as interfaces between the various pipeline stages). Instructions stall for different number of cycles in various latches between the pipeline stages, which increases the probability of an error in that particular latch and hence in that particular active structure. When considering pipeline stalls, λ_{A_j} has to be multiplied by the average number of cycles an instruction stalls in the pipeline stage A_j. We denote this number of cycles by X. Hence, the instruction error rate (which we denote by *MIER*) now becomes:

$$MIER = \sum_{i=1}^{n} \frac{\lambda_{R_i}}{S_i} C_i + \sum_{j=1}^{k} \frac{\lambda_{A_j}}{W_j} X_j \quad (5)$$

In equation (5), C_i denotes the average number of cycles that an entry of the storage structure R_i is in use (averaged over the entire program execution). Similarly, X_j denotes the average number of cycles that an instruction stalls in a pipeline stage (again, averaged over the entire program execution). Shivakumar, et. al. [14] predict that the contribution of the static logic may soon equal that of latches, in which case, equation (5) will have to be modified.

Different instructions access different data storage structures. For instance, some instructions use the branch predictor, whereas the others do not. Similarly, some instructions may use the register file and the others may get their values from data forwarding or as an immediate value. Hence, we weigh each data storage structure (R_i) (in equation (5)) by the percentage of instructions (out of the total committed) that access that structure (P_{Ri}). P_{Ri} also takes into account the cases where an instruction may have multiple unique accesses to the structure (for instance, an instruction reading 2 different registers). However, we assume that all instructions go through all the pipeline stages, even if an instruction does not use a pipeline stage (*e.g.* non-memory instructions need not go through the memory-accessing pipeline stages). Note that we use the percentage of instructions out of the total committed because an error in

an instruction along a mispredicted path will not result in a wrong program outcome. Hence, *MIER* is given by:

$$MIER = (\sum_{i=1}^{TR} \frac{\lambda_{R_i}}{S_i} C_i P_{Ri} + \sum_{j=1}^{PS} \frac{\lambda_{A_j}}{W_j} X_j) \quad (6)$$

Equation (6) gives the instruction error rate only for instructions along the correct path of execution, if one instruction is executed per cycle. Since multiple instructions go through the pipelines simultaneously in a superscalar processor, the instruction error rate of equation (6) is multiplied by the average number of useful instructions (those that commit) executing simultaneously (*i. e.* IPC). Hence, *MIER* is given by:

$$MIER = (\sum_{i=1}^{TR} \frac{\lambda_{R_i}}{S_i} C_i P_{Ri} + \sum_{j=1}^{PS} \frac{\lambda_{A_j}}{W_j} X_j) \times IPC \quad (7)$$

In equations (6) and (7), TR gives the total number of data storage structures in the processor, and PS gives the total number of pipeline stages. Note that there may be some instructions that are committed but do not affect the program outcome, such as those that write a register which is never used. An error in such an instruction is included in the *MIER* computed using equation (7), because such instructions do contribute to the instruction failure rate. If such errors are not to be considered, equation (7) can be multiplied by the percentage of instructions that do affect the program outcome. Similarly, branch instructions may use wrong values but result in correct branch outcome, thus not affecting the program execution. *MIER* in equation (7) also includes such branch instructions.

The transient error model given by equation (7) has a limitation that the values of parameters such as P_{Ri}, X_j, C_i, and IPC have to be measured using simulations to get an accurate $MIER$. In addition, the model assumes that the MTBFs of the hardware modules remains constant, whereas, they could change with changes in operating conditions such as temperature.

2.2 Experimental Model

We also propose a detailed experimental model where errors are induced while simulating benchmarks on a high performance microprocessor simulator. In our experiments, errors are generated in the various hardware entities, based on the *MTBF* of the entity. To generate an error in our experimental transient error model, we generate a set of random numbers (one for each of the data storage structures and the active structures, as described in the previous section) every cycle. For each of the data storage structures in the processor, we generate an error in one of the bits of one of the entries of that structure based on the MTBF and the random number generated for that structure. To decide the entry and the bit in that entry where an error is induced, another set of random numbers are generated. The data storage entry is then marked as faulty. Whenever an instruction accesses a particular storage structure, and an error is encountered in the entry accessed, the instruction is also marked as faulty. Similar process is used to generate an active error in one of the active structures (pipeline stages). The random number generated and the MTBFs for the *active structures* determine whether an error will occur in any of the active structures. If an error occurs in an active structure, then based on more random number generations, an error is induced in one of the pipelines in that particular active structure, and if an instruction is executing on that pipeline, it is marked as faulty. If there are no instructions executing on the pipeline where the error is induced, then no errors occur. When an instruction commits, it is checked for an error. The number of failed committed instructions and the total number of cycles spent in execution are counted to determine *MIER*.

2.3 Component MTBF

The MTBF of any hardware module depends on the radiation flux rate and the underlying circuit error rate. Radiation flux rate results from the contamination in the packaging material, and the terrestrial neutrons, and can depend on the geographical location of the chip. The circuit error rate depends on the technology and the implementation of the circuit. Overall, the soft error rate of a hardware module is generally estimated using radiation testing and circuit simulation tools [5]. However, all the soft errors do not affect the program outcome. For instance, errors in the static logic may not propagate to the forward latch (*e. g.* if the error does not occur within the setup and hold time of the forward latch) [14], thus not affecting the program outcome. This can be taken into consideration by multiplying the soft error rate of a circuit cell by its timing vulnerability factor [5, 12].

Next, we corroborate the instruction error rate obtained from the analytical model with that obtained from the experimental model.

Parameter	Value	Parameter	Value
Fetch/Decode/ Commit Width	8 instructions	*FP FUs*	3 ALU, 1 Mul/Div
Unified Phy. Register File	128 INT/128 FP entries, 2-cycle acc. lat. 1-cycle inter-subsystem lat.	*Int. FUs*	4 ALU, 2 AGU 1 Mul/Div
		ROB Size	256
Issue Width	5/3 INT/FP instructions	*Issue Queue*	96 INT/64 FP Instructions
Branch Predictor	Gshare 4K entries	*BTB Size*	4K entries, 2-way assoc.
L1 - I-cache	32K, direct-map, 2 cycle latency	*L1 - D-cache*	32K, 4-way assoc., 2 cycle latency, 2 r/w ports
Memory Latency	100 cycles first word 2 cycle/inter-word	*L2 - cache*	unified 512K, 8-way assoc., 10 cycles

Table 1: Baseline Processor Hardware Parameters for the Experimental Evaluation

3 Experimental Results

3.1 Experimental Setup

The hardware parameters for the base superscalar processor are given in Table 1. 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. Our base pipeline consists of 9 front-end stages. 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.

In our transient error models, the data storage structures that we consider are the ROB, the integer and floating point register files, the rename map table, the load/store queue, the branch predictor, and the issue queue. As discussed in Section 2.3, the component MTBFs are computed using extensive testing. In our experiments, we assume different values of MTBFs for the components to corroborate the analytical model. To obtain *MIER* from the analytical model, we run the simulations to get the parameters used in equation (7). In the experimental model, the random numbers generated in our experiments have a very large period, approximately $16 * (2^{31} - 1)$, which ensures that the pattern of random numbers generated repeats for a different set of instructions. We observed that even though the pattern of random numbers repeated, there never was a case where the pattern of errors repeated. For instance, if in one set of random numbers an error was induced in the 10th bit of the 20th register in the integer register file, the same error was not repeated when the same pattern of random numbers were encountered.

3.2 Experimental Results

We measure *MIER* in terms of failed instructions per cycle for both the analytical and the experimental transient error model. For this, we use *MTBFs* of the various hardware entities in terms of number of cycles between failures. In our experiments, we use the same *MTBF* for all the hardware (data storage as well as active) structures. Figure 1 shows the average (over all the benchmarks studied) percentage difference of *MIER* obtained from the analytical model with respect to the experimental model, as *MTBFs* for the hardware structures is varied from 1×10^7 cycles to 100 cycles. We do not choose values greater than 1×10^7 cycles because some hundred million cycles are only required to execute the 1B instructions.

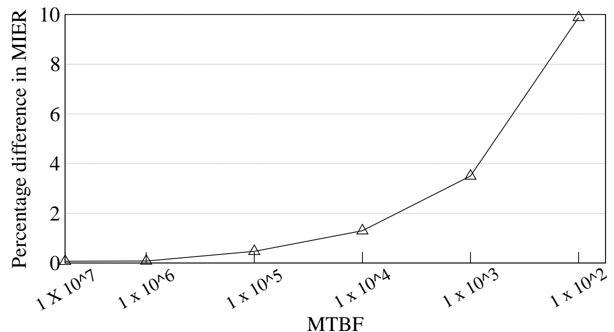

Figure 1: Average percentage difference in *MIER* obtained from analytical model wrt that obtained from experimental model

Figure 1 shows that the *MIER* obtained from the analytical model is very close to that obtained from the experimental model. This is especially true for larger *MTBFs*. For lower *MTBFs*, the percentage difference between the analytical and the experimen-

tal model increases, with the analytical model always giving more number of errors than the experimental model. This discrepancy between the analytical and the experimental models results because the analytical model assumes that each hardware error results in an error in a unique instruction. In practice, this may not be true for low *MTBFs*, where multiple errors may be encountered by a single instructions and the experimental model will count such an instruction only once. As *MTBF* decreases, the number of such instances, where the analytical model computes multiple errors and the experimental model just one, increases. Our experiments show that *MIER* obtained from the analytical model reaches almost 10% more than that obtained from the experimental model for an *MTBF* of 100 cycles. We repeated the experiments for various hardware configurations, and observed very similar results. For instance, we measured the $MIER$ with both the analytical and experimental models as the pipeline depth and width are changed and the sizes of the different storage structures are changed, and found that the *MIER* obtained from the analytical model closely followed that obtained from the experimental model.

We also measured the difference in *MIER* between the analytical and the experimental model, when considering errors in only the data storage structures and when considering errors in only the active structures. Figure 2 presents the results. As seen in Figure 2, the difference in *MIER* is more for the active structures, because the probability of an instruction encountering multiple errors due to active structures is more than that due to data storage structures. An instruction only accesses certain entries in the various data storage structures, and for an instruction to encounter multiple errors, multiple entries accessed by the instruction must be faulty, which should intuitively have a low probability. This is especially true when the storage structures have a large number of entries and the entries get over-written very quickly. However, as an instruction flows down a pipeline, there is a higher probability that an error occurs in the pipeline used by the instruction in multiple pipeline stages, especially if the instruction stalls for a number of cycles in the various pipeline stages.

4 Performance with Errors

To measure the performance of a system in the presence of errors, we require the total number of errors encountered and the penalty (in terms of the number of cycles) for each error. If I is the total number of instructions committed, and E is the total number of

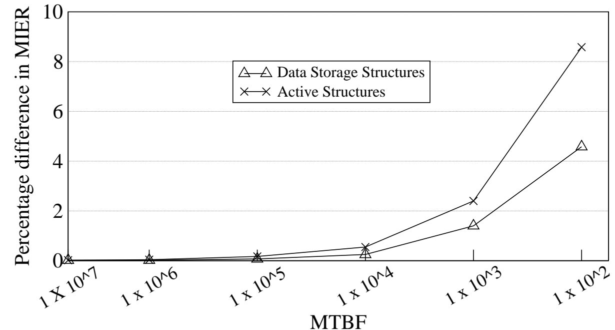

Figure 2: Average percentage difference in *MIER* obtained from analytical model wrt that obtained from experimental model, for just the **data storage structures** and **active structures**

errors encountered, then E is given by:

$$E = MIER \times Executiontime = Z \times I \quad (8)$$

where, $Z = (\sum_{i=1}^{TR} \frac{\lambda_{R_i}}{S_i} C_i P_{Ri} + \sum_{j=1}^{PS} \frac{\lambda_{A_j}}{W_j} X_j)$

If the execution cycles of an application (without any errors) is given by E_{ne}, then the execution cycles with errors will be given by $E_{ne} + E \times N$, where N is the error penalty in terms of the number of cycles wasted when an error is encountered. In most of the error detection and recovery techniques [9, 10, 11, 16, 4], where multiple copies of the same instructions are executed and their results are corroborated at the commit time, error penalty can be considered to be equal to the pipeline depth plus the number of cycles taken to recover from the error.

5 Impact of Hardware Parameters on Errors

For any hardware modifications to give performance improvement, the improvement in execution time due to the modification should overweigh the impact of the modification on the number of errors and the cycles wasted for an error. In this section, we discuss the impact of various hardware parameters on the number of errors encountered, and hence on the performance of the system. The first parameter that we consider is the pipeline depth. An increase in the pipeline depth will increase the error rate of the active structures, mainly because of an increase in the number of pipeline stages PS, which increases the number of latches used in the microprocessor. P_{Ri} will generally remain constant with an increase in pipelining. Increasing pipeline depth will also increase C_i because the values will remain in the data storage

structures for more number of cycles. Similarly, increasing pipeline depth may also increase X_j. Hence, an increase in pipeline depth will increase the number of errors, and may also increase the error penalty.

Next, we consider speculative hardwares (such as a data value predictor) that may be used to improve the performance of a microprocessor. These hardware structures usually consist of a predictor table, used for making the predictions. With an addition of speculative hardwares, a change in the number of errors depends on the change in C_i, X_j, and λ_{A_j}. All the other parameters in equation (8) are expected to remain the same. Note that we do not include the error rate in the predictor table in determining the change in the number of errors, because any errors in the predictor table will affect the speculation accuracy, and hence C_i and X_j. Therefore, the error rate of the predictor table will be reflected in the change in C_i and X_j. The error rate of the active structure in the pipeline stage in which the speculative hardware is accessed may increase due to the addition of the predictor. However, the major impact of speculative hardware will be on C_i and X_j, where C_i and X_j may reduce due to correct speculations (because of a faster passage of the instructions through the pipeline), and increase due to misspeculations (because of misspeculation penalties). Overall, any speculative technique with high prediction accuracy can result in a reduction in C_i and X_j, thereby reducing the number of errors and improving performance.

Next, we consider modifications to the size of non-speculative data storage elements (such as the register files and ROB). Size modifications of these elements will affect the number of errors by affecting C_i, X_j, and λ_{A_j}. We assume that the error rate per entry of the storage element does not change with a change in the size of the element (because of a corresponding increase in λ_{R_i}). λ_{A_j} is affected because some modifications may be required in the active structures in the pipeline stages where the data storage elements are accessed. However, as in the case of speculative hardware, non-speculative storage elements will have a major impact only on C_i and X_j. An increase in the size of such elements will generally decrease C_i and X_j, because of faster execution of instructions. Overall, it is expected that the number of errors will reduce as the size of the non-speculative data storage elements is increased, resulting in a better performance.

Our experiments with various hardware configurations confirmed the analysis. We do not present the results here to conserve space.

6 Comparative Studies

In this section, we compare the various error detection and recovery techniques proposed in the literature, based on the analytical model of Section 2.1. First, we compare the single-chip simultaneous execution (*SME*) [9] and single-chip slacked execution (*SSE*) [10, 11, 16, 4] error detection and recovery techniques. In *SME*, an application is simultaneously run multiple times on the same chip and the results of the various runs are corroborated for error detection. In *SSE*, an application is still run multiple times on the same chip, however, the replicated execution thread runs at a slack (of a few instructions) with respect to the original execution thread. All the parameters of equation (8) remain same for both the *SSE* and *SME* techniques, except C_i and X_j, provided the hardware used for the both the approaches is the same. There may be some differences in the hardware, because *SME* uses a replication hardware, and may also use another pipeline stage for replication, whereas *SSE* may use multiple fetch units for the multiple threads.

C_i and X_j are lower for the *SSE* technique than for the *SME* technique because of the slack between the original and the replica thread. For instance, whenever the replica thread is stalled to maintain the slack, most of the resources are available to the original thread, and it runs faster. When the original thread is stalled (maybe due to branch misprediction or load miss), most of the resources are available to the replica thread making it run faster. In addition, the replica thread may not incur the penalties due to load misses and branch mispredictions, if the load miss has been serviced and the branch has been evaluated for the original thread. Hence, the instructions tend to flow through the pipeline faster, if any of the thread is stalled. For cases where both the original and the replica thread are not stalled, C_i and X_j for the *SSE* technique are similar to that of the *SME* technique, and for the cases where either of the threads is stalled, C_i and X_j for the *SSE* technique are lower than the *SME* technique because the instructions issue faster, the register values and the maps get used faster, and instructions (especially the replica instructions) get committed soon after they are dispatched to the ROB. This suggests that the errors encountered in the *SSE* technique will be lower than that encountered in the *SME* technique. Studies [16, 4] have already shown that the *SSE* technique is expected to perform better than the *SME* technique, and combined with the low occurrence of errors in the *SSE* technique, its overall performance is expected to be better than that of the *SME* technique.

Next, we compare the *SSE* and the multi-chip simultaneous execution (*MSE*) error detection and recovery techniques. In the *MSE* technique, multiple chips execute the same application simultaneously. If the errors in the multiple chips are assumed to be independent, then the total number of errors in the *MSE* technique will be almost equal to twice that given by equation (8). However, the number of errors encountered in the *SSE* technique will be more than double of that given by equation (8). This is because, compared to an execution of a single thread, double the number of instructions are committed for the *SSE* technique, and C_i and X_j are also higher. However, the overall comparison between *MSE* and *SSE* would also depend on the number of cycles required for recovering from an error, as *MSE* requires off-chip communications for detection and recovery of errors. If the error rate is lower, and the off-chip communication can be performed off-line, then the *MSE* technique is expected to perform better than the *SSE* technique, which is expected because of the additional hardware utilized in the *MSE* technique.

7 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, 15, 16, 4]. 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, 16, 4] 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.

As far as modeling the instruction error rate for performing the trade-offs between performance and reliability, there has only been one other attempt by Weaver, et. al. [17]. Weaver also realized that need to transform the hardware error rate into instruction error rate. Their model to define mean instruction to failure (*MITF*), although similar to ours, is at a much coarser granularity than ours. They define *MITF* in terms of the raw error rate of the entire processor, the IPC, the frequency of the processor, and the architectural vulnerability factor [6] of the entire processor. The architectural vulnerability of a structure depends on the number of cycles that various values spend in that structure (such as C_i and X_j in our model). Our approach derives the instruction error rate using the combined effect of the error rate of each hardware structure inside the processor. Considering the entire processor may give a skewed instruction error rate for the following reasons: (i) not all the structures are used equally by the instructions, and the error rate of each structure should be weighed by its usage, (ii) the soft error rate of a hardware structure may vary from another on the same die due to process variations, and (iii) the architectural vulnerability of a hardware module will differ from another. In addition, if processors are built with different power supplies and clock frequencies for the different modules, then considering the entire processor may not even be practical.

8 Conclusion

With the current trends in transistor size, voltage, and clock frequency, microprocessors are becoming increasingly susceptible to transient (soft) failures. Traditionally, soft error rates have been defined in terms of the rate at which the hardware generates erroneous results due to faults. However, this definition does not fit a high performance microprocessor model, consisting of various hardware modules and executing instructions. This is because a transient error in a hardware module does not necessarily generate a fault in program execution, because the faulty hardware may not be in use when an error occurs. In addition, the faulty instruction resulting from the faulty hardware may not contribute to the final result of the program. Hence, it is imperative to transform hardware error rates into instruction error rates that can be used in a high performance processor.

In this paper, we present detailed analytical and experimental transient error models to transform the hardware error rate into instruction error rate. We introduced a *mean instruction error rate (MIER)* metric that gives failed instructions per unit time. The models employ a bottom-up approach, where they decompose the hardware into separate entities, each with its own mean time between failures. The models then combine the hardware error rates of the different entities to derive *MIER*. When deriving *MIER*, the models also consider the architectural issues such as the number of cycles the values and instructions are alive in a pipeline. The instruction error rates obtained from the two models were found to be very consistent.

We then apply the analytical model to study how

different hardware parameters affect the number of errors encountered in a processor. We also apply the model to compare the performance of various error detection and recovery techniques.

References

[1] T. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," *Proc. Micro-32*, 1999.

[2] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," *Computer Arch. News*, 1997.

[3] Compaq Computer Corp., "Data integrity for Compaq Non-Stop Himalaya servers," *http://nonstop.compaq.com*, 1999.

[4] M. Gomaa, et. al., "Transient-Fault Recovery for Chip Multiprocessors," *Proc. ISCA-30*, 2003.

[5] S. Mitra, et. al., "Robust System Design with Built-In Soft-Error Resilience," *IEEE Computer*, Vol. 38, No. 2, Feb. 2005.

[6] S. Mukherjee, et. al., "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," *Proc. Micro-36*, 2003.

[7] S. Mukherjee, et. al., "The Soft Error Problem: An Architectural Perspective," *Proc. HPCA-11*, 2005.

[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] N. Seifert and N. Tam, "Timing Vulnerability Factor of Sequentials," *IEEE Trans. Device and Materials Reliability*, Vol. 4, No. 3, Sep. 2004.

[13] D. P. Siewiorek and R. S. Swarz, "Reliable Computer Systems Design and Evaluation," *The Digital Press*, 1992.

[14] 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.

[15] K. Sundaramoorthy, et. al., "Slipstream processors: Improving both performance and fault tolerance," *Proc. Micro-33*, 2000.

[16] T. Vijaykumar, et. al., "Transient-fault recovery using simultaneous multithreading," *Proc. ISCA-29*, 2002.

[17] C. Weaver, et. al., "Techniques to Reduce the Soft Error Rate of a High Performance Microprocessor," *Proc. ISCA-31*, 2004.