On the Correlation between Controller Faults and Instruction-Level Errors in Modern Microprocessors*

Naghmeh Karimi†	Michail Maniatakos	Abhijit Jas	Yiorgos Makris
ECE Department	EE Department	Validation & Test Solutions	EE & CS Departments
University of Tehran	Yale University	Intel Corporation	Yale University

Abstract

We investigate the correlation between register transfer-level faults in the control logic of a modern microprocessor and their instruction-level impact on the execution flow of typical programs. Such information can prove immensely useful in accurately assessing and prioritizing faults with regards to their criticality, as well as commensurately allocating resources to enhance testability, diagnosability, manufacturability and reliability. To this end, we developed an extensive infrastructure which allows injection of stuck-at faults and transient errors of arbitrary starting point and duration, as well as cost-effective simulation and classification of their repercussions into various instruction-level error types. As a test vehicle for our study, we employ a superscalar, dynamically-scheduled, out-of-order, Alpha-like microprocessor, on which we execute SPEC2000 integer benchmarks. Extensive experimentation with faults injected in control logic modules of this microprocessor reveals interesting trends and results, corroborating the utility of this simulation infrastructure and motivating its further development and application to various tasks related to robust design.

1. Introduction

As aggressive scaling continues to push technology into smaller feature sizes, various design robustness concerns continue to arise. Among them, the frequent occurrence of transient errors has resurfaced as a contemporary problem of interest. Part of the problem is attributed to strikes by neutrons or alpha particles and the corresponding single event upsets (SEUs) in memory bits, or single event transients (SETs) in combinational logic, which may potentially result in a soft error. As we move forward, however, errors occurring due to various other issues related to design marginalities, process variations and corner operating conditions are starting to play an equally important role. Interestingly, such errors may range in duration from single events to permanent faults. As a result, interest in concurrent error detection (CED) and/or correction methods that may ameliorate or resolve their effect has been revived.

CED is a topic that has been extensively studied in the past [1, 2, 3, 4, 5]. However, while a plethora of solutions exist, blindly applying them across the board is, most likely, both prohibitive in terms of cost and unnecessary in terms of the attained coverage. Indeed, not all transient errors incur the same level of criticality and not all protection mechanisms actually contribute to the overall robustness of a design. Therefore, methods to analyze the relative importance of potential transient errors and the relative effectiveness of candidate countermeasures are invaluable for developing cost-effective solutions.

Such analysis becomes even harder in modern microprocessors, wherein advanced architectural features complicate the process. It comes, thus, as no surprise that existing transient error analysis methods focus either at the microscopic circuit level [6, 7, 8, 9], or at the macroscopic architectural level [10, 11, 12], but typically refrain from attempting a bridging between the two. Inevitably, however, such bridging mechanisms that can assess the criticality of circuit-level faults by correlating them to their impact on application-level functionality are the only way to justify the cost of any method added to suppress them. And while a few efforts [13, 14, 15, 16] have been expended in the past to establish such correlations, they concern earlier generations of microprocessors that do not comprise the latest architectural features (e.g. out-of-order execution, dynamic scheduling, mutiple issue, etc.). The latter, however, constitute the main source of complexity in modern microprocessors, thus limiting the applicability of these previous efforts.

The objective of the research described in this paper is to develop the infrastructure necessary for exploring the correlations between low-level control logic faults and their impact on instruction-level execution of typical programs running on modern high-performance microprocessors. While the datapath of such microprocessors is equally important, we mainly focus on their control logic for the following two reasons. First, CED for datapath is understood much better and various coding techniques have been successfully applied. Second, advanced architectural features complicate significantly the task of the controller, making it much harder to analyze or predict its behavior in the presence of low-level errors.

The starting point for developing the aforementioned infrastructure is a public-domain high-performance microprocessor, which is briefly discussed in Section 2. Section 3 presents the various components and capabilities of the developed infrastructure, along with the flow of its multi-level utilization for fault injection, simulation and correlation. An instruction-level error model is introduced in Section 4; this model is then used to demonstrate the correlation capabilities of the infrastructure through extensive experimental results provided in Section 5. Finally, our plans for extending the developed infrastructure and investigating its applicability and effectiveness in guiding the use of CED methods and other test and reliability related tasks are discussed in Section 6.

*Research supported through a generous gift from Intel Corporation.

†Research performed while author was a visiting student at Yale University.

1-4244-4203-0/08/$20.00

©2008 IEEE

2. Test Vehicle: The IVM Processor

We start by briefly presenting the microprocessor that we will use as the test vehicle in our investigation. We discuss the capabilities of the simulation infrastructure that has been previously developed by other researchers around this microprocessor, we pinpoint its limitations and we identify its components that need to be enhanced in order to support our study.

2.1. Microprocessor Model and Functional Simulator

Since the focus of this work is the cross-correlation between control logic defects and instruction level errors in modern microprocessors, the underlying test vehicle should incorporate as many of the state-of-the-art architectural features as possible. Among the very limited number of such test-cases available in the public domain, we chose to work with a Verilog implementation of an Alpha-like microprocessor, called IVM (Illinois Verilog Model) [17]. IVM implements a subset of the instruction set of the Alpha 21264 microprocessor. Consisting of approximately 40,000 state elements, the IVM is rich in architectural features including: superscalar, out-of-order execution, dynamically scheduled pipeline, hybrid branch prediction and speculative instruction execution. IVM can have up to 132 instructions in-flight through its 12-stage pipeline, supported by a dynamic scheduler of 32 entries and 6 functional units. The complexity of IVM reflects most of the features of modern, high-performance microprocessors; thus, it enables a realistic investigation of the instruction-level impact of control logic errors in such microprocessors. Besides the Verilog implementation, a functional simulator that can be used in conjunction with the IVM processor model also exists. This functional simulator simulates the full set of the Alpha 21264 microprocessor and is part of the SimpleScalar tool suite implemented for the Multiscalar Research Project [18].

2.2. Capabilities and Limitations

The IVM microprocessor was developed and used to study the impact of single-event transient errors IVM [17, 19, 20], modeled as single register-level bit-flips. Unfortunately, gate-level fault simulation cannot be performed; due to certain coding techniques used at the RT-level model, the current version of IVM is not synthesizable. Instead, an approach of stopping the simulation, altering the state of the microprocessor, and then continuing the simulation is currently employed. This fault injection approach is effective when studying the impact of single-cycle transient errors, such as those by alpha particle strikes. However, it is extremely inefficient for other fault models, such as stuck-at faults or transient errors of longer duration caused by operational marginalities. Indeed, the process of injecting a fault for a clock cycle involves extensive file-system-based interactions and becomes very time consuming if done for more than a few clock cycles. To alleviate this limitation, we develop an enhanced infrastructure which supports efficient fault injection and simulation for these longer-lasting errors, as described in Section 3.

Another key aspect of the existing infrastructure is that both IVM and the functional simulator can execute SPEC benchmarks. This is important since it allows us to study the impact of errors while the microprocessor is executing a typical workload, thus making our findings more realistic. However, the IVM does not support the full instruction set of Alpha; floating point instructions and various system-calls have not been implemented. Therefore, the functional simulator must be used to surmount this limitation, by invoking it whenever such instructions need to be executed. This interaction is enabled through the ability of the functional simulator to load/store the state of the Verilog model and vice-versa at any given clock cycle.

3. Enhanced Simulation Infrastructure

We now proceed to describe the fault simulation enhancements that we developed based on the aforementioned infrastructure, as well as the pertinent tool-flow that enables our error correlation investigation. We first outline the main capabilities of the enhanced infrastructure, followed by a detailed description of its basic components and a discussion of its utilization.

3.1. Capabilities

Starting with the Alpha-like IVM microprocessor model described in Section 2, we augment it to provide the additional capabilities necessary for the correlation study targeted by our investigation. Overall, the enhanced infrastructure provides the following key features:

- **Simulation:** We can simulate software written for the Alpha microprocessor. In our experiments, we use SPEC2000 Integer benchmarks.
- **Fault injection & simulation:** We can perform fault injection into any register or combinational logic entity instantiated at the RT-level of the microprocessor by augmenting the model accordingly. Fault injection is controlled by a fault controller module inside the microprocessor. The fault simulation infrastructure supports both stuck-at faults and transient error models of used-defined starting times and durations.
- **Trace dumping:** The model can produce traces for all the modules of the microprocessor for specified clock cycles.
- **State dumping:** At any given clock cycle, we can save all information concerning microprocessor state, such as the contents of SRAMs, flip-flops, register files etc.

3.2. Main Components

The enhanced fault simulation infrastructure consists of three main parts, as shown in Figure 1: i) supporting tools to control fault injection and the I/O of the procedure, ii) a functional simulator of the Alpha microprocessor, and iii) the augmented version of the microprocessor model. The main functionality of the

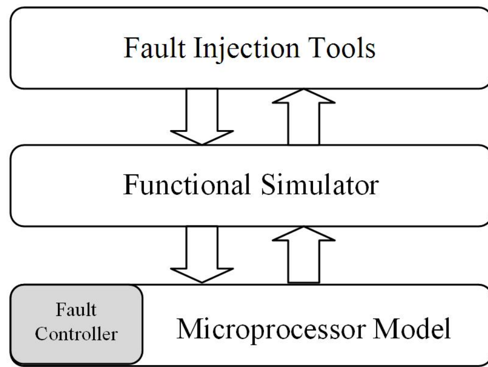

Figure 1. Infrastructure Components and Interactions

fault injection tools is to provide support to the functional simulator by generating the appropriate files and passing parameters for specific operations. Also, these tools accumulate the results of the simulations and report them to the user.

3.2.1. Functional Simulator

The presence of a functional simulator in the flow is essential because it enhances the functionality of the microprocessor model. Since the current version of IVM does not implement floating point operations, system calls and miscellaneous other instructions of the Alpha 21264 processor, these instructions can be executed via the the functional simulator which implements the complete instruction set. The simulation can be switched from the functional simulator to the Verilog model and vice versa at any given time. In practice, for the SPEC2000 Integer Benchmarks used in this study, the functional simulator is used to skip the initial system calls, after which the execution continues at the RT-level model of the IVM microprocessor.

Furthermore, the functional simulator enhances the I/O functionality of the Verilog model. Thus, it can read values from files and pass parameters to the Verilog model during transition between states. It can also output the state or traces of the microprocessor model to the file system. These features enable fault injection and analysis as well as trace dumping through the developed simulation infrastructure.

3.2.2. Microprocessor Model

Since the existing version of the IVM microprocessor model cannot be synthesized so that gate-level fault injection and simulation can be performed, an alternative efficient way for doing this is required. To this end, we augment the IVM model so as to support fault injection capabilities at the RT-Level. Specifically, a Fault Controller module is added to the microprocessor, controlling the fault injection process. When this module is deactivated, the microprocessor operates normally as a fault-free circuit. When it is activated, it provides an extensive range of options for injecting faults. Since the module is already built in the microprocessor model, consecutive simulations injecting different faults can be executed without recompiling the model, something that would make any reasonably-sized fault simulation experiment computationally prohibitive. Besides the insertion of a new module, each existing module of the IVM microprocessor is also augmented to provide support for the functions of the Fault Controller, as explained in detail below.

Table 1. Input/Output Interface of Fault Controller

Type	Name	Bits
Input	fault_index	32
Input	fault_bit_index	8
Input	fault_type	2
Input	error_cycle_start	32
Input	error_cycle_end	32
Output	fault_register	42
Output	fault_clock	1

3.2.3. Fault Controller

The main component of fault injection is the embedded Fault Controller module. This module is only activated during fault injection cycles, if any; otherwise it does not interfere with the execution flow of the microprocessor. Similarly to any other module, the Fault Controller has a specified list of inputs and outputs, as presented in Table 1.

The registered inputs of the Fault Controller are not connected to and do not interact with the microprocessor model; instead, the functional simulator is responsible for setting these registers to the appropriate values. The output of the Fault Controller propagates to all modules of the microprocessor. In addition, the Fault Controller outputs a clock, which specifies whether a fault should be injected. Specifically, at the rising edge of this clock each module receives a signal indicating that a fault injection should occur, prompting the module to process the outputs of the Fault Controller and behave accordingly.

During the simulation, the Fault Controller is responsible for fault injection. In each clock cycle, we can access one bit of one entity and set it to a specific value, where the entity can be either a register or a wire. We call this procedure *Fault Addressing*; every entity of the microprocessor is assigned a unique identification number (UID), which can be any arbitrarily chosen number. When the Fault Controller activates the fault clock, each module compares the broadcasted UID to the UIDs of its internal entities. If a match is found, the module modifies the corresponding bit, as specified by the outputs of the Fault Controller that are sent to the module. This fault injection technique is similar to the parallel saboteurs injection technique [21]. An extensive comparison to existing fault injection approaches can be found in [22].

For a module to be able to respond to the Fault Controller functions, it must be augmented accordingly. For this purpose, after assigning a UID to each entity, a piece of code that will enable Fault Addressing within each module must be generated. Moreover, a fault list containing all faults for every bit of each entity must also be generated. Both fault code and fault list generation are performed by internally developed fault injection tools.

After describing the main mechanism underlying the fault injection process, we now list the usage of the various registers in the Fault Controller:

- `fault_index`: Specifies the UID of the entity to be fault-injected. If the UID is invalid, no entity will be fault-injected. Similarly, if more than one entity share the same UID, they will all be fault-injected.
- `fault_bit_index`: Specifies the bit index of the UID to be fault-injected.
- `fault_type`: Specifies the type of the injected fault. Our infrastructure supports stuck-at faults and transient errors of specified duration.
- `error_cycle_start`: Specifies the clock cycle at which the fault injection should commence.
- `error_cycle_end`: Specifies the clock cycle at which the fault injection should terminate.
- `fault_register`: Contains all the necessary information that should be passed to the modules (i.e. the UID of the entity, the bit index of the entity and the fault type to be fault-injected).
- `fault_clock`: The clock that activates fault injection within the modules.

In this way, by manipulating the data stored in the registers, we can perform single-cycle transient error injection, duration-controlled transient-error injection, or stuck-at fault injection. Furthermore, we can perform periodic transient error injection by continuously updating the registers that contain the fault injection starting and stopping cycles.

3.2.4. State and Trace Files

The presence of the functional simulator presence in the infrastructure augments the I/O capabilities of the RTL model. Developed infrastructure enables the generation of files that store the state of the microprocessor and files that store the I/O of modules at any given clock cycle. Similarly to [20], information regarding the states of all flip-flops and SRAMs in the microprocessor, including the register file and the main memory, as well as information regarding the memory structures supporting the simulation model can be obtained. More specifically, at any given clock cycle, we can stop the simulation and save the current state. This is crucial for fault simulation since we can first create these state traces for a golden run where no faults are injected and then, during fault simulation, compare them to the state trace of a fault-injected run in order to assess the impact of the fault on the microprocessor.

Besides state files, we can also log the inputs and the outputs of any given module at specified clock cycles, producing a trace file. This file can then be used to study the impact analysis of errors at individual modules. Furthermore, the trace file provides useful statistics about the activation and usage of the I/O of a module, such as identifying the most frequently used wires, their switching frequency etc.

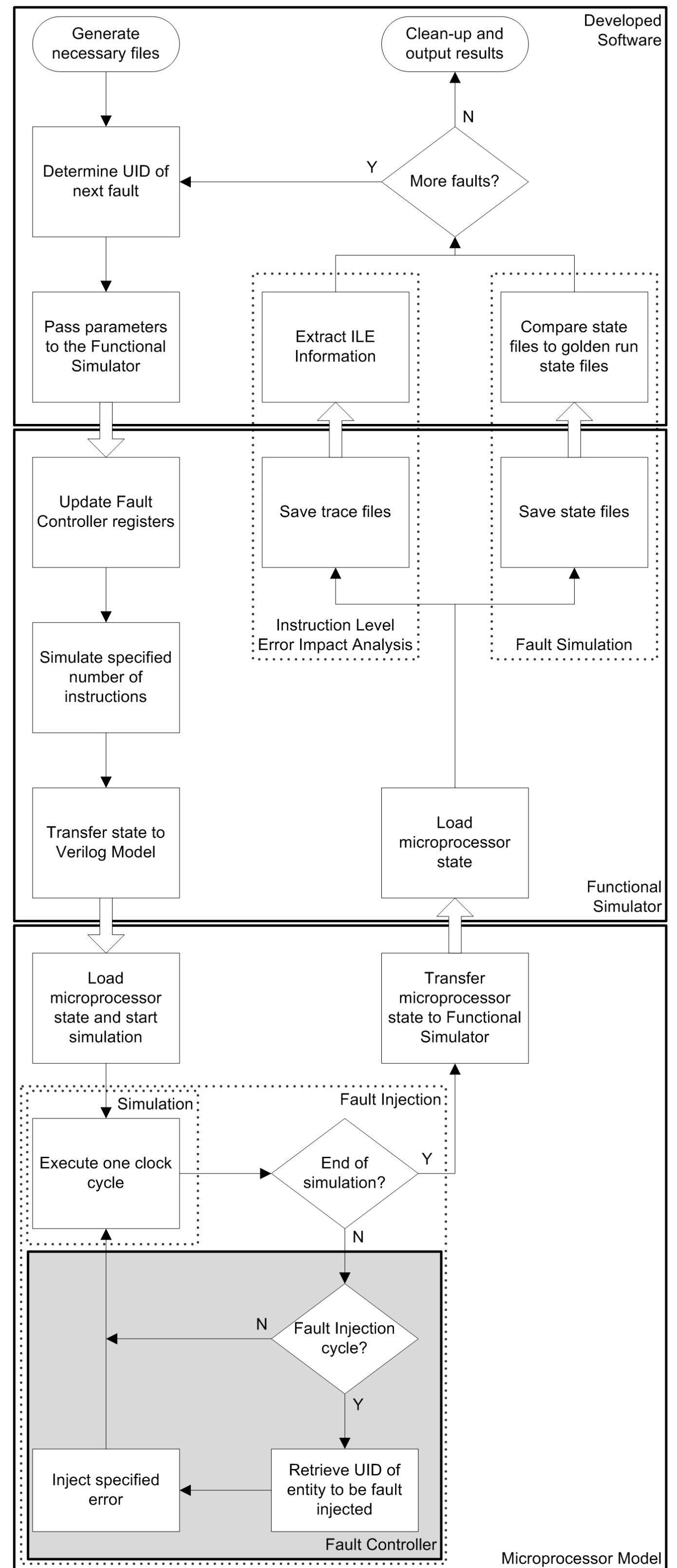

Figure 2. Infrastructure Utilization Flow

3.3. Simulation Flow

After presenting each component of the infrastructure, we now describe how all components are combined to provide the aforementioned capabilities. Figure 2 presents a flow chart of the procedure, where each of the three distinct components of Figure 1 and their interactions are now depicted in more detail.

Initially, for each entity that will be fault-injected, the corresponding fault code and fault list should be present in the fault-injection model. Fault injection tools initialize the procedure, parse the given fault list and produce the necessary files to guide the functional simulator.

Following the initialization phase, the functional simulator starts execution and parses the fault injection parameters while updating the Fault Controller registers. Once the values are correctly set-up, the functional simulator executes a user-specified number of instructions. Given the fact that IVM lacks system call support, initial system calls requested by applications must be executed at the functional simulator. When the simulator completes execution, the microprocessor state is transferred to the Verilog model.

The Verilog implementation of the Alpha microprocessor simulates the rest of the program code. After a user-specified number of clock cycles, which is provided through a register in the Fault Controller, the latter activates the fault clock through which it instructs all modules to check whether they should alter any included entity (i.e. perform fault injection during the next clock cycle). At the end of the simulation the state is saved and transferred back to the functional simulator. Note that the functional simulator doesn't simulate the effects of faults unless they've propagated to memory or output.

The functional simulator simply outputs the data collected by the Verilog model and stops execution. Subsequently, fault injection tools collect the data and perform the operations requested by the user. We should note that the whole process is very flexible and parameterized; the user can choose functionality (e.g. trace dumping, fault analysis), which faults to inject, when to inject each fault, and how long to inject it for. In this way, various types of studies are facilitated.

The impact of mutation on simulation performance is negligible in a huge design of a modern microprocessor and can be further leveraged by selective mutation. The simulation speed is limited by the size and complexity of the Verilog implementation.

4. Instruction-Level Errors (ILEs)

We continue by introducing various types of instruction-level errors (ILEs), organized in several groups. While these ILE types constitute neither a complete nor a mutually exclusively set, they have been carefully selected to reflect incorrect behavior occurring due to errors in the control logic of a modern microprocessor. Thereby, these ILE types enable us to study the cross-correlation between defects in the low-level hardware implementation of control logic and their high-level impact from the point of view of instruction execution.

4.1. ILE Groups & Types

In this study, we consider thirteen types of ILEs, organized in five distinct groups, as summarized in Table 2. Such grouping reflects the five key aspects of instruction execution in a super-scalar out-of-order microprocessor, namely (i) the operation that is executed, (ii) the operands that are being used, (iii) the functional unit where execution takes place, (iv) the starting and finishing time of execution, and (v) the order of commitment. The various ILE groups and types are discussed in more detail below.

4.1.1. Group 1: Operation Errors

The first group covers errors in the sequence of instructions executed by the microprocessor, as reflected through discrepancies in their operation code (op_code). Such discrepancies can cause one of the following ILE types:

- **Type 1:** The op_code of an instruction is mutated to another op_code that is valid but incorrect.
- **Type 2:** The op_code of an instruction is changed to an invalid op_code.

4.1.2. Group 2: Operand Errors

The second group covers errors in the operands that are being used by an instruction. In certain instructions, such errors may also affect the instruction execution flow of a program and can, therefore, be considered as control errors. Our error model covers both register and immediate operands through the following ILE types:

- **Type 3:** A register address used by an instruction points to a valid but incorrect location in the register file.
- **Type 4:** A register address used by an instruction points to an invalid location in the register file.
- **Type 5:** An instruction uses the contents of a register prematurely, essentially violating a Read-After-Write (RAW) constraint.
- **Type 6:** An instruction uses an incorrect immediate value as one of its operands.

4.1.3. Group 3: Execution Errors

Super-scalar microprocessors employ several functional units of various types (e.g. integer ALUs, floating point ALUs, branch unit, memory operation unit, etc.), in order to execute multiple instructions simultaneously. Accordingly, the third group covers errors in the utilization of these functional units by the executed instruction through the following ILE types:

- **Type 7:** An instruction is assigned to and executed by a functional unit of incorrect type.
- **Type 8:** An instruction is assigned to more than one functional unit.

Table 2. Instruction-Level Errors

Group		Type	
Group 1:	Operation Errors	**Type 1:**	Incorrect (yet valid) op_code used
		Type 2:	Invalid op_code used
Group 2:	Operand Errors	**Type 3:**	Incorrect (yet valid) register addressed
		Type 4:	Invalid register addressed
		Type 5:	Premature use of register contents
		Type 6:	Incorrect immediate operand used
Group 3:	Execution Errors	**Type 7:**	Incorrect functional unit type utilized
		Type 8:	Multiple functional units utilized
Group 4:	Timing Errors	**Type 9:**	Early commencement
		Type 10:	Late or no commencement
		Type 11:	Longer duration
		Type 12:	Shorter duration
Group 5:	Order Errors	**Type 13:**	Commitment order violation

Table 3. Information Collected to Support ILE Classification

Fields Logged for Each Instruction Executing in Each Functional Unit at Each Clock Cycle							
Instruction Robid	Instruction Op_code	Register Operands	Availability of Register Operands	Immediate Operand	ID of Executing Functional Unit	Starting Time of Execution	Ending Time of Execution

4.1.4. Group 4: Timing Errors

The fourth group covers discrepancies in the timing of instruction execution. Such discrepancies manifest themselves via incorrect starting and/or finishing instruction time-stamps and are captured through the following ILE types:

- **Type 9:** An instruction commences execution at an earlier clock-cycle than it is supposed to.
- **Type 10:** An instruction commences execution at a later clock-cycle than it is supposed to, or does not commence execution at all.
- **Type 11:** An instruction completes execution in a longer period of time than it is supposed to.
- **Type 12:** An instruction completes execution in a shorter period of time that it is supposed to.

4.1.5. Group 5: Order Errors

The fifth group covers errors in the order in which instructions are executed and committed. In a processor with out-of-order execution capabilities, instructions can be scheduled and executed out of order. Therefore, a reorder buffer is typically used to keep track of the instructions that are in flight and ensure that they are committed in order. Errors causing discrepancies in this order are captured by the following ILE type:

- **Type 13:** The correct order of instruction commitment is violated.

4.2. ILE Classification Process

In order to be able to appropriately categorize the impact of a low-level control logic discrepancy into one of the ILE types introduced in the previous section, we use our fault simulation infrastructure to collect the necessary information. More specifically, for each clock cycle, we log various fields related to the execution of instructions in the processor. This is first done for a golden run, wherein no fault is injected, and subsequently for the fault-injected processor. The two traces of information are compared and, at the first point of failure, the corresponding fields are used to classify the injected fault into an ILE type. The information that is collected during each clock cycle is summarized in Table 3:

1. the op_code of the instruction being executed; this is simply the type of the instruction. Based on this, ILEs of Types 1-2 can be identified.
2. the physical addresses of the source and destination registers that are used by the instruction; this shows where the operands reside and where the result will be written. Based on this, ILEs of Types 3-4 can be identified.
3. the ready bits of these registers; this indicates whether the source operands are ready to be read. Based on this, ILEs of Type 5 can be identified.
4. the values of any immediate operands than the instruction may be utilizing. Based on this, ILEs of Type 6 can be identified.
5. the identification number of the functional unit where the instruction is executed. Based on this, ILEs of Types 7-8 can be identified.
6. the clock-cycle at which the instruction starts execution. Based on this, ILEs of Types 9-10 can be identified.
7. the clock cycle at which the instruction is expected to finish execution. Based on this, ILEs of Types 11-12 can be identified.
8. the robid of the instruction being executed; this is an identification number assigned by the Reorder Buffer which follows the instruction until it commits and serves as the

mechanism for ensuring in-order instruction commitment in the out-of-order execution IVM microprocessor. Based on this, ILEs of Type 13 can be identified.

4.3. Example

As an example of ILE classification, let us consider Timing ILEs of Types 11-12. We should clarify that, given the scope of the paper, the term Timing Error refers to errors that affect the duration or the commencement/retirement of an instruction, and are different from timing errors at lower levels of the design. Assume that in the fault-free case, an instruction starts executing at time t_1 and finishes at time t_2. However, in the presence of a fault, this instruction starts at t_1 but finishes at t_3, $t_3 \neq t_2$. The occurred fault is either a long duration ILE (Type 11) or a short duration ILE (Type 12). To classify this ILE, we use of the fields in the last two columns of Table 3, namely the starting time and finishing times of execution of the golden and the faulty model. Certain timing errors can be benign, but in the scope of our study we still consider them as ILEs.

5. Experiments

In this section, we demonstrate the error cross-correlation capabilities and the corresponding insight that can be gained through the developed infrastructure. To this end, we perform a series of simulations wherein RT-level faults are injected in the control logic of the IVM microprocessor while the latter executes SPEC benchmarks and we analyze their instruction-level impact. We first discuss the details of the fault simulation setup; then, we present the obtained results and we reflect on the information that they provide and its potential significance in various aspects of microprocessor design and test.

5.1. Fault Simulation Setup

While, ultimately, we envision the use of our infrastructure for studying error correlations across multiple abstraction levels, for the purpose of this experiment we examine the cross-correlation between the RT-level and the instruction level. In addition, since the focus of this work is on control logic errors, we concentrate on two of the key control modules of the IVM microprocessor, namely the Scheduler and the Reorder Buffer (ROB). Within these two modules, we choose a fault model that includes all stuck-at faults in registers. Performance errors (like branch prediction errors) are not considered in this study, as we focus on errors that corrupt the program output, even though some errors within the out of order logic may result as performance errors.

The total number of faults injected is 27,234, covering both stuck-at-0 and stuck-at-1 faults. No fault collapsing or fault dropping techniques were used.

With regards to the applications executed on the IVM microprocessor during fault simulation, seven standard SPEC2000 Integer benchmark programs are used. In each fault simulation run, the functional simulator is used to execute the first 50,000 clock cycles, thus bypassing the initial system calls and other operations not implemented in IVM. Subsequently, the microprocessor state is passed to the Verilog implementation of the model, which executes another 2,000 clock cycles, during which the injected fault is active. This process mimics the occurrence of a transient error of long duration at some random point during program execution. The use of 7 different benchmarks ensures variability of the instruction flow executed in the 2,000 clock cycle window. Because of the continuous nature of the injected stuck-at faults, errors at the instruction level either appear very early in the execution or are not activated at all. The latter happens either due to fault masking or (mostly) because certain register bits are rarely used in a typical execution flow (e.g. the most significant bits of address registers, or scheduler slots that are used only when a fairly large number of instructions are in flight). Correlation between controller faults and Instruction-Level Errors (ILEs) is independent of the fault model used; results showed that a fault at a bit results in the same ILE in all different benchmarks. Thus, the use of stuck-at fault model maximizes the probability of an ILE appearing in the simulation window.

When a fault-simulation using a benchmark completes the 2,000 cycle window, its trace is compared line-by-line to the trace of the golden run; the comparison is performed by automated tools. In the event of a discrepancy, comparison tools utilize certain checks and algorithms to classify the fault to the appropriate Instruction-Level Error (ILE) type, as described in Section 4.2. Even though we compare line-by-line for differences between the golden trace and the faulty trace, when a discrepancy is found, information from multiple cycles is used to correctly classify the error. However, only the first discrepancy is reported and classified as an ILE, because the execution after that point is corrupted and will result in many different ILEs. If more than one ILE are identified in the clock cycle of first ILE appearance, all of them are reported.

As a first set of results, we present cumulative data regarding the fault simulations performed. Specifically, Table 4 reports the percentage of the 27,236 injected faults that resulted in an ILE, as well as the average number of ILE types that are simultaneously activated for each of the seven SPEC2000 benchmark programs that were executed. Based on this table, the following observations can be made:

- The number of faults resulting in an ILE ranges between 24%-40%. Intuitively, faults injected during the execution of benchmark programs using a limited variety and algorithmic combination of instructions will excite fewer ILEs due to a larger portion of unused processor functionality.
- Since the ILE types are not mutually exclusive, more than one ILE types may be activated simultaneously, even when checking in a cycle-by-cycle fashion. However, as can be observed in the last column of Table 4, the average number of simultaneously activated ILE types is only 1.27; this implies that, most of the time, only one of the 13 ILE types is activated at the first point of failure. The implication of this information is that corruption will typically occur only at one aspect of instruction execution, with the rest remain-

Table 4. Results on SPEC2000 Integer Benchmarks

SPEC Benchmark Name	Simulations Resulting in ILEs	%	Average # of ILEs Activated Simultaneously
bzip2	10935	40%	1.33
cc	10511	39%	1.34
gap	6548	24%	1.20
gzip	7263	27%	1.10
mcf	6929	25%	1.26
parser	10927	40%	1.37
vortex	10743	39%	1.31
Average	9122	33%	1.27

ing unaffected. Thus, early detection and identification of such ILEs can guide simple operations towards restoring the correct state of the microprocessor.

Our second set of results examines the consistency of the information provided through our experiments. Specifically, since each benchmark utilizes different functional capabilities of the processor, the ILE type resulting from a stuck-at fault may vary, depending on the actual instructions being executed. In this sense, the robustness of the extracted cross-correlation information may be questioned. Therefore, in Figure 3, we present the percentage of stuck-at faults that results in ILEs of each of the five groups described in Section 4.1, for each of the seven benchmark programs. Based on this bar-chart, the following observations can be made:

- The distribution of stuck-at faults to the five groups of ILEs is consistent across the seven benchmarks. Furthermore, the variance of stuck-at faults within each group across the seven benchmarks is small[1]. These observations corroborate that the obtained cross-correlation information is not biased by the actual instructions executed by each benchmark program and is, therefore, robust.
- A large percentage of stuck-at faults (60%-80%) result in timing errors, implying that most stuck-at faults in the control logic may not affect the instruction itself but, rather, when this instruction is executed. This is expected since faults injected in the Scheduler and the ROB modules directly impact instruction issuing and commitment. Such information is very useful in guiding allocation of error detection and recovery resources. In this case, for example, one would focus on methods that predict and monitor the correctness of instruction starting and stopping times, since, thereby, the majority of the faults would be detected.

5.2. Results & Discussion

The third set of presented results relates to the occurrence frequency of each of the 13 ILE types described in Section 4.1.

[1]We note that these observations hold even when the groups are further broken down to the constituent ILE types. For the sake of clarity, the corresponding figure is omitted.

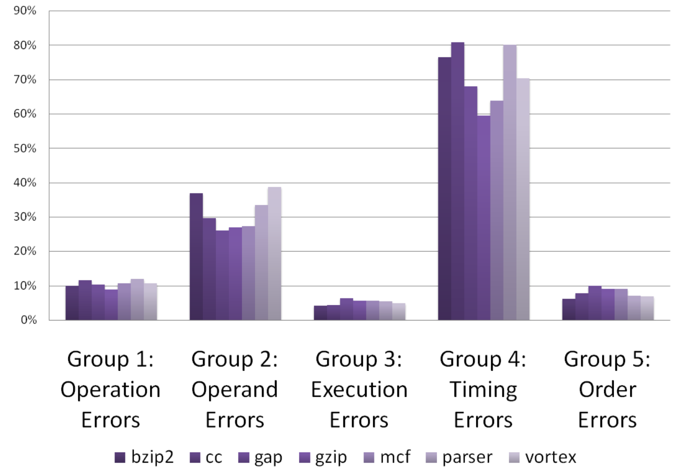

Figure 3. Percentage of Stuck-at Faults Causing each ILE Group for each of the Seven Benchmarks

The average number of stuck-at faults resulting in each ILE type over the seven benchmarks is presented in Figure 4. The subset of these faults that eventually result in stalling of the pipeline is also provided. The following observations can be made based on the results:

- The most frequently occurring ILEs concern instruction execution timing issues. Specifically, late instruction commencement (Type 10) and longer instruction duration (Type 11) are the dominant types. In other words, faults injected in the Scheduler and the ROB module will often result in failure to issue an instruction or failure to commit an instruction at the appropriate clock-cycle. Another interesting observation is the frequent appearance of operand mutations (Type 3). Indeed, the complex structures employed by the scheduler to keep track of the 1 to 3 registers used by each instruction, appear to be vulnerable to various stuck-at faults in the control logic.
- On the other hand, stuck-at faults in the Scheduler and the ROB seem to rarely cause mutation of operation codes (Type 2) or invalid operand address (Type 4), since the logic involved is relatively limited. Similarly, very few faults cause premature use of operands (Type 5), incorrect functional unit assignment (Types 7-8), or out-of-order instruction commitment (Type 13). In these cases, the involved logic can be large, but its complexity is such that it prevents single stuck-at faults in a register from activating these ILE types, hence their low occurrence probability.

The insight provided by the aforementioned observations regarding the most frequently occurring and, thus, the most critical ILE types can be leveraged to facilitate cost-effective use of error detection and correction resources. by implementing small, efficient CED techniques targeting a specific ILE type

Another very interesting set of results pertains to the time that elapses between injection of a fault and its appearance as one of the defined ILE types, as well as the latency between appearance of an ILE and a potential subsequent stalling of the

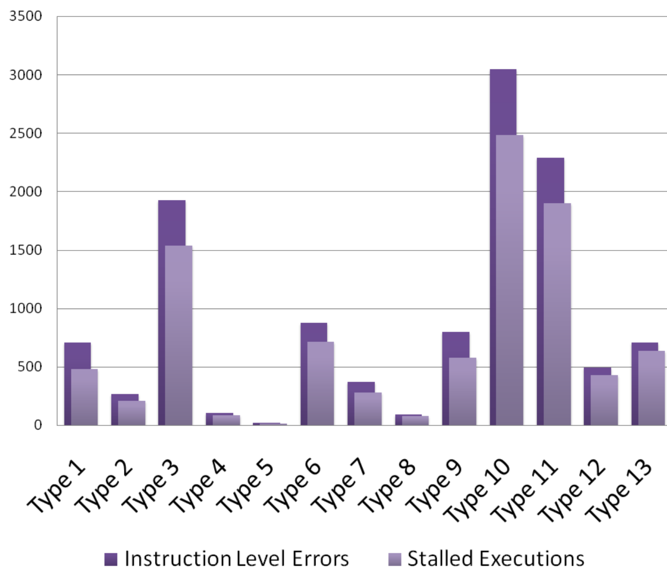

Figure 4. Average Number of Stuck-at Faults Causing each ILE Type and Subsets Causing Stalled Execution

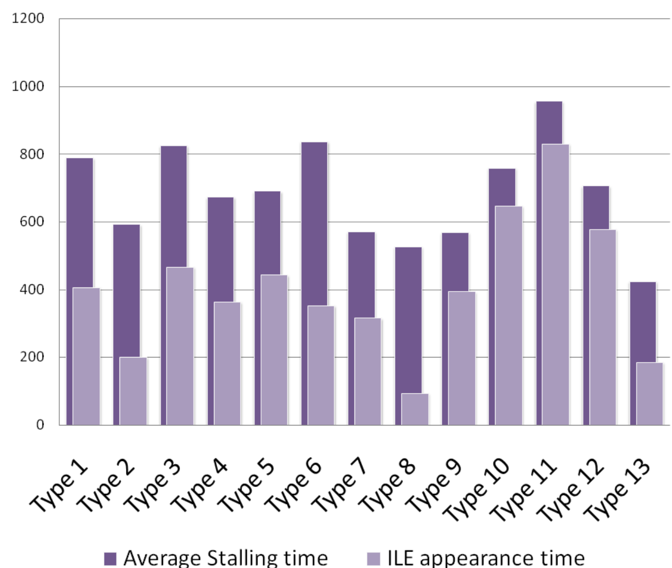

Figure 5. Average Time-stamp of ILE Identification and Subsequent Pipeline Stalling (in clock cycles)

microprocessor. This information is provided in Figure 5 for the fault simulations performed in our experiments. While the activation time of an ILE may depend on the actual sequence of executed instructions, averaging the results over the various SPEC benchmarks provides an unbiased estimate. The obtained results motivate the following observations:

- The average time until an injected fault results in an ILE is 406 clock cycles. However, the standard deviation across the 13 ILE types is rather high (198 clock cycles), with some ILE types occurring very quickly and others much later. For example, invalid operation code (Type 2), utilization of multiple functional units (Type 8) and incorrect commitment order (Type 13) are examples of ILEs appearing very quickly after fault injection. Indeed, despite the fact that the set of faults causing these ILEs is relatively small (as presented in Figure 4), such faults are directly associated with these specific types of ILEs. Thus, upon the appearance of such a fault, the corresponding ILE is immediately activated. On the other hand, ILEs related to timing issues (e.g. Types 9-11) appear much later. Such ILEs seem to be often the result of faults propagating to parts of the microprocessor that do not interfere directly with the main pipeline flow, yet eventually work their way into it and, hence, the longer latency.
- On average, a microprocessor stall occurs 280 clock cycles after occurrence of an ILE. However, one may observe that some ILEs concerning timing issues (i.e. Types 9-11) result in microprocessor stalling much faster than the rest of the ILEs; given the fact that these ILEs are caused when the Scheduler or the ROB fail to timely issue or commit an instruction, subsequent instructions fail to issue or commit successfully, inevitably causing the pipeline to stall very quickly. On the other hand, for ILEs such as the Type 7 discussed above, stalling appears much later, after the Scheduler and/or the ROB fill up with instructions waiting for the incorrectly executed instruction to retire.

  The insight provided by the aforementioned observations is two-fold. First, they reveal the relative temporal criticality of each ILE type in terms of Mean Time To Failure (MTTF). Thus, they can be used to fine-tune error tolerance methods that employ checkpoints to examine and restore the microprocessor state [19]. Second, they indicate the window of opportunity for correcting an error before it drastically corrupts the processor state and results in a stall. Thus, they can be leveraged to prioritize the allocation of error detection and correction resources.
- Finally, Figure 5 shows that the activated ILEs appear no later than 800 cycles after fault injection; this ensures that the window of 2,000 clock cycles that we observe is sufficiently long for the performed analysis. Still, there are ILEs activated later in the simulation which are masked in the results shown in Table 4. Given that their percentage is relatively small, the simulation window is limited to 2,000 to increase performance.

6. Future Directions

The introduced infrastructure provides the foundation for investigating various robust design issues in modern microprocessors. As a natural next step, we are currently developing a synthesizable version of the IVM microprocessor model in order to extend our correlation capabilities across the gate-level, the RT-level and the instruction-level. At the same time, we are leveraging the existing infrastructure in order to assess the effectiveness of existing concurrent error detection and correction methods, as well as to develop new ones. Furthermore, we plan to expand the scope of our analysis to all control modules of the microprocessor, thus obtaining a global picture of its most

vulnerable parts. Finally, we anticipate that extensive correlation information between gate-level faults and instruction-level errors can also enhance complex design diagnosis and support debugging efforts, which we intend to explore further.

7. Conclusion

The analysis described in this paper is neccessary to successfully develop optimal designs conerning CED techniques. In order to perform this analysis, an infrastructure with extensive capabilities has to be build. Such capabilities are provided by the infrastructure described herein, which enables injection and simulation of RT-level faults in an Alpha-like microprocessor and correlation of their impact to instruction-level errors in SPEC2000 integer benchmarks. The insight obtained through this multi-level cause and effect analysis proves instrumental to the efficient utilization of resources. Indeed, as we demonstrated, crucial observations can be made regarding the activation frequency and latency of various instruction-level errors, thus guiding the selection of appropriate concurrent error detection and correction methods. For example, extensive experiments demonstrated that most low-level faults cause timing issues in the instruction execution flow. Furthermore, latency fluctuates throughout the different error types, providing valuable information on error type prioritization. Similar insight and guidance for various other design robustness endeavors can be achieved through the application of the existing infrastructure and the described extensions.

Acknowledgement

The authors would like to thank Prof. Sanjay Patel and Nicholas Wang from the University of Illinois at Urbana-Champaign for sharing the IVM microprocessor model and for providing technical assistance in its installation and usage.

References

[1] M. Goessel and S. Graf, *Error Detection Circuits*, McGraw-Hill, 1993.

[2] C. Metra, M. Favalli, and B. Ricco, "On-line detection of logic errors due to crosstalk, delay, and transient faults," in *ITC*, 1998, pp. 524–533.

[3] S. Mitra and E. J. McCluskey, "Which concurrent error detection scheme to choose?," in *International Test Conference*, 2000, pp. 985–994.

[4] K. Mohanram and N. A. Touba, "Cost-effective approach for reducing soft error rate in logic circuits," in *ITC*, 2003, pp. 893–901.

[5] S. Almukhaizim, P. Drineas, and Y. Makris, "Entropy-driven parity-tree selection for low-overhead concurrent error detection in finite state machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 8, pp. 1547–1554, 2006.

[6] Cha Hungse, E.M. Rudnick, J.H. Patel, R.K. Iyer, and G.S. Choi, "A gate-level simulation environment for alpha-particle-induced transient faults," *IEEE Transactions on Computers*, vol. 45, no. 11, pp. 1248–1256, 1996.

[7] T. M. Austin, "Diva: A reliable substrate for deep submicron microarchitecture design," in *International Symposium On Microarchitecture*, 1999, pp. 196–207.

[8] N. Seifert, Zhu Xiaowei, and L.W. Massengill, "Impact of scaling on soft-error rates in commercial microprocessors," *IEEE Transactions on Nuclear Science*, vol. 49, no. 6, pp. 3100–3106, 2002.

[9] T. Karnik, P. Hazucha, and J. Patel, "Characterization of soft errors caused by single event upsets in cmos processes," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 128–143, 2004.

[10] A. Mendelson and N. Suri, "Designing high-performance and reliable superscalar architectures - the out of order reliable superscalar (o3rs) approach," in *International Conference on Dependable Systems and Networks*, 2000, pp. 5–28.

[11] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *IEEE/ACM International Symposium on Microarchitecture*, 2003, pp. 29–40.

[12] G. Weining, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, "Characterization of linux kernel behavior under errors," *IEEE Transactions on Dependable and Secure Computing*, vol. 00, pp. 459, 2003.

[13] E.W Czeck and D.P Siewiorek, "Effects of transient gate-level faults on program behavior," in *International Symposium on Fault-Tolerant Computing*, 1990, pp. 236–243.

[14] M. Rimen and J. Ohlsson, "A study of the error behavior of a 32-bit risc subjected to simulated transient fault injection," in *International Test Conference*, 1992, pp. 696–.

[15] G. Miremadi and J. Torin, "Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection," *IEEE Transactions on Reliability*, vol. 44, no. 3, pp. 441–454, 1995.

[16] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: a flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 248–260, 1995.

[17] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ACE analysis reliability estimates using fault-injection," *SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 460–469, 2007.

[18] D. Burger, T. M. Austin, and S. Bennett, "Evaluating future microprocessors: The simplescalar tool set," Tech. Rep. CS-TR-1996-1308, 1996.

[19] N. J. Wang and S. J. Patel, "Restore: symptom based soft error detection in microprocessors," in *International Conference on Dependable Systems and Networks*, 2005, pp. 30–39.

[20] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *International Conference on Dependable Systems and Networks*, 2004, pp. 61–70.

[21] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into vhdl models: the mefisto tool," *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pp. 66–75, Jun 1994.

[22] J. C. Baraza, J. Gracia, S. Blanc, D. Gil, and P. J. Gil, "Enhancement of fault injection techniques based on the modification of vhdl code," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 6, pp. 693–706, June 2008.