Instruction-Level Impact Analysis of Low-Level Faults in a Modern Microprocessor Controller

Michail Maniatakos, *Student Member*, *IEEE*, Naghmeh Karimi, *Student Member*, *IEEE*, Chandrasekharan (Chandra) Tirumurti, *Member*, *IEEE*, Abhijit Jas, *Member*, *IEEE*, and Yiorgos Makris, *Senior Member*, *IEEE*

Abstract—We investigate the correlation between low-level faults in the control logic of a modern microprocessor and their instruction-level impact on the execution of typical workload. 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 online testability and error/fault resilience through concurrent error detection/correction methods. To this end, we developed an extensive fault simulation infrastructure which allows injection of stuck-at faults and transient errors of arbitrary starting time 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 fault injection campaigns in control modules of this microprocessor facilitate valuable observations regarding the distribution of low-level faults into the instruction-level error types that they cause. Experimentation with both Register Transfer (RT-) and Gate-Level faults, as well as with both stuck-at faults and transient errors, confirms the validity and corroborates the utility of these observations.

Index Terms—Fault simulation, instruction-level error, microprocessor controller, concurrent error detection.

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. Notably, such errors may range in duration from single events to permanent faults. As a result, interest in enhancing online testability and error/fault resilience through concurrent error detection (CED) and/or correction methods has been revived.

- *M. Maniatakos is with the Department of Electrical Engineering, Yale University, New Haven, CT 06520-8267. E-mail: michail.maniatakos@yale.edu.*
- *N. Karimi is with the Department of Electrical Engineering, Duke University, Durham, NC 27708. E-mail: naghmeh.karimi@duke.edu.*
- *C. Tirumurti is with the Validation and Test Solutions Group, Intel Corporation, Santa Clara, CA 95050. E-mail: chandra.tirumurti@intel.com.*
- *A. Jas is with the Validation and Test Solutions Group, Intel Corporation, Austin, TX 78746. E-mail: abhijit.jas@intel.com.*
- *Y. Makris is with the Department of Electrical Engineering, The University of Texas at Dallas, Richardson, TX 75080-3021. E-mail: yiorgos.makris@utdallas.edu.*

Manuscript received 5 June 2009; revised 21 Oct. 2009; accepted 17 Dec. 2009; published online 23 Feb. 2010.
Recommended for acceptance by C. Metra and R. Galivanche.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2009-06-0258.
Digital Object Identifier no. 10.1109/TC.2010.60.

While a plethora of CED solutions have been developed in the past [1], [2], [3], [4], [5], [6], [7], blindly applying them across the board is not only prohibitive in terms of cost but also unnecessary in terms of the attained coverage. Indeed, not all faults incur the same level of criticality and not all protection mechanisms contribute equally to the overall robustness of a design. Therefore, methods which analyze the relative importance of potential faults and the relative effectiveness of candidate countermeasures are invaluable for developing cost-effective solutions.

Modern microprocessors, in particular, exhibit an inherent effectiveness in suppressing a significant percentage of faults and preventing them from interfering with correct program execution (i.e., application-level masking). In other words, the probability that a fault will adversely impact the typical workload of a microprocessor varies greatly, depending on the frequency with which the corresponding hardware is used and the complexity of the control conditions necessary to propagate its effect to the architectural state of the microprocessor. Hence, application-level masking presents a great opportunity for developing cost-effective CED methods by identifying and targeting the most critical faults.

To this end, the research described in this paper seeks to provide the ability to assess the relative importance of low-level faults (i.e., faults in the RT- or Gate-Level description) in the control logic of a modern high-performance microprocessor, as gauged by their impact on the execution of typical programs. Specifically, the contributions of this paper include:

- An extensive infrastructure built around a modern microprocessor model, enabling simulation of low-level faults and analysis of their instruction-level impact during execution of typical workload.

- An instruction-level error model, reflecting the key aspects of instruction execution, to which the impact of low-level faults is mapped.
- A comprehensive set of fault simulation results demonstrating the correlation between low-level faults in a modern microprocessor controller[1] and the instruction-level errors that they incur.

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 its utilization flow for fault injection, simulation, and impact analysis. The proposed instruction-level error model, which is used to capture the impact of low-level faults on program execution is introduced in Section 4. Extensive fault simulation campaigns using the developed infrastructure are presented in Section 5, along with a detailed analysis of the obtained results and a discussion of their significance in guiding the development of cost-effective CED methods.

2 Test Vehicle

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 faults 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 Illinois Verilog Model (IVM) [8]. IVM implements a subset of the instruction set of the Alpha 21264 microprocessor, and 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 six 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 faults in such microprocessors. Besides the Verilog implementation, a functional simulator that can be used in conjunction with the IVM processor model through the Verilog Procedural Interface (VPI) also exists. This functional simulator supports the full set of the Alpha 21264 processor and is part of the SimpleScalar tool suite implemented for the Multiscalar Research Project [9].

1. While the data path of such microprocessors is equally important, we mainly focus on their control logic for the following two reasons. First, CED for data path 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 faults.

2.2 Capabilities and Limitations

IVM was developed and used to study the impact of single-event transient errors [8], [10], [11], 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, IVM is not synthesizable. Instead, an approach of stopping the simulation, altering the state of the microprocessor, and then resuming the simulation was employed in these studies. This fault injection approach is effective when studying the impact of single-cycle transient errors, such as those caused 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 enhanced this infrastructure to support efficient injection and simulation for such longer-lasting faults, as we describe in Section 3.

Another key aspect of the existing infrastructure is that both IVM and the functional simulator can execute software compiled for the Alpha microprocessor. This is important, since it allows us to study the impact of faults while the microprocessor is executing a typical workload, thus making our findings more realistic. However, 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 added to the aforementioned infrastructure, as well as the pertinent tool-flow that enables our study. 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

We augmented the IVM microprocessor model described in Section 2 to provide the following key features:

- **Workload 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 wire entity of the microprocessor by mutating the model accordingly. Fault injection is controlled by a fault controller module inside the microprocessor. The fault simulation infrastructure supports stuck-at faults and transient errors of user-specified start time and duration.
- **Trace dumping:** The model can produce traces at the periphery of any module of the microprocessor for any user-specified number of clock cycles.

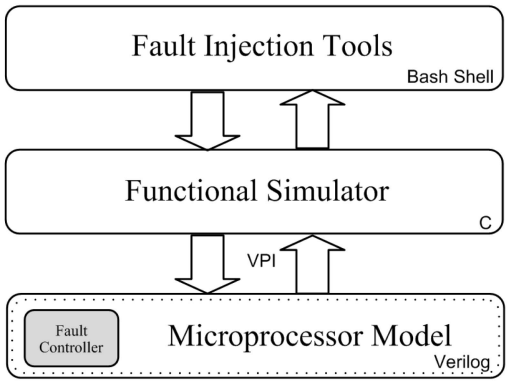

Fig. 1. Infrastructure components and interactions.

- **State dumping:** At any given clock cycle, we can save all information concerning the machine state, as well as the architectural state of the processor.
- **Incorporation of Gate-Level modules:** Synthesizable versions of two modules, namely the Scheduler and the ReOrder Buffer (ROB), have been implemented and can substitute their RT-Level counterparts in the microprocessor model.

3.2 Main Components

The enhanced fault simulation infrastructure consists of three main parts, as shown in Fig. 1: 1) supporting tools to control fault injection and the I/O of the simulation procedure, 2) a functional simulator of the Alpha microprocessor, and 3) the augmented version of the microprocessor model, where a Fault Controller module has been added to enable Fault Injection.

3.2.1 Fault Injection Tools

The main functionality of the fault injection tools is to provide support to the functional simulator by generating the appropriate files and passing parameters for specific operations (e.g., which fault location to inject, the type of injected fault, etc.), as well as accumulating and reporting the simulation results to the user.

Furthermore, the developed tools can be used to save any trace or state files requested and perform comparisons between golden (fault free) and faulty model executions. The state files contain information regarding the states of all flip-flops and SRAMs in the microprocessor, including the register file and the main memory. The user can choose to extract any subset of the aforementioned storage elements (e.g., only the architectural register file). 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 of faults on 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.

3.2.2 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 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, using VPI calls. 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 enables reading of values from files and passing 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.

3.2.3 Microprocessor Model

The microprocessor model used is IVM, which was briefly described in Section 2.1. Since the existing IVM version cannot be synthesized so that gate-level fault injection and simulation can be performed, an alternative way for doing this is required. Even if the complete processor was synthesizable, fault-simulating the entire Gate-Level model would probably be impractical. Hence, we are limited to using RT-Level logic simulation tools, such as Synopsys VCS. This enables simulation of the RT-Level model of the microprocessor while the latter executes actual workload, but it does not offer fault injection capabilities.

To address this limitation, we mutate the IVM model so as to support RT-Level fault injection. 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, however, 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 IVM is also mutated to provide support for the functions of the Fault Controller, as explained in detail in Section 3.2.5.

3.2.4 Fault Controller

The main component of fault injection is the embedded Fault Controller module, whose list of inputs and outputs is presented in Table 1 and described below:

- fault_index: Specifies the unique identification number (UID) of the entity to be fault injected. Every entity in the microprocessor is assigned an UID. If the UID is invalid, no entity will 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.
- `error_cycle_start` : Specifies the clock cycle at which the fault injection should commence.

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

- `error_cycle_end` : Specifies the clock cycle at which the fault injection should terminate.
- fault_register: Outputs all the information that should be passed to the modules (i.e., fault_index, fault_bit_index, and fault_type).
- `fault_clock` : The clock that activates fault injection within the modules.

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. 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, when this clock is 1 each module receives a signal indicating that a fault injection should occur, prompting the module to process the outputs of the Fault Controller.

3.2.5 Fault Injection

During 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*. 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 [12]. An extensive comparison to existing fault injection approaches can be found in [13].

For a module to be able to respond to Fault Controller functions, it must be mutated accordingly. For this purpose, after assigning an 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. Depending on whether a module is described as a netlist or as behavioral Verilog, the module is mutated differently:

RT-Level fault injection. For behavioral Verilog, injection is performed in every entity defined in the Verilog model. Fig. 2 presents a simplified diagram of the method, which is capable of injecting either stuck-at faults or transient faults, with user-defined starting and stopping times (dotted, lighter colored lines indicate resources added for fault simulation purposes). Since we operate at the RT-Level model, only entities described in the Verilog model are fault injected. Each entity is driven by a `MUX`, which is controlled by the Fault Controller. The *fault clock* signal alters the value of the target entity during the active fault injection window. Each entity has a unique ID, so that the Fault Controller can pick the one to be injected in each clock cycle.

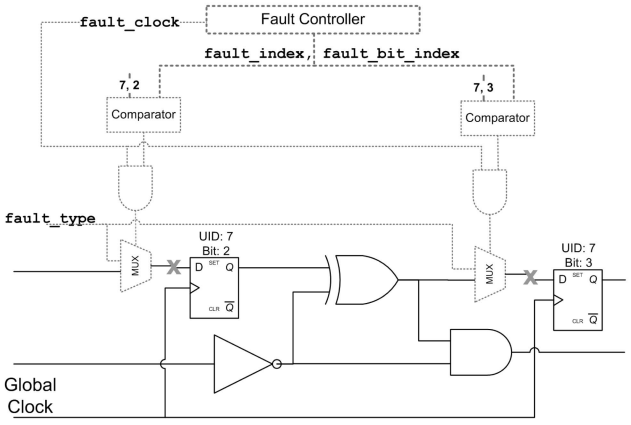

Fig. 2. Fault injection in latches of RT-Level model.

Gate-Level fault injection. Netlist fault injection can also be performed during RT-Level simulation, thereby supporting Gate-Level fault simulation. A simplified version of the method is depicted in Fig. 3 (dotted, lighter colored lines indicate resources added for fault simulation purposes). In order to avoid cluttering the figure, we only show the resources for three out of the 10 fault injection sites; the rest of the sites are injected in the exact same way. In this simulation environment, every wire has an additional driver, which is controlled by the Fault Controller. Cells are treated as black boxes. The Fault Controller drives a high-impedance value `z` whenever a wire should not be fault injected, so that the normal value of the wire is propagated. During fault injection, a `supply0` or a `supply1` value is driven on the wire, resembling a stuck-at-0 or a stuck-at-1 fault, respectively. According to Verilog definition, if a wire has multiple drivers the strongest one prevails, with `z` being a neutral value. `Supply0` and `supply1` are the strongest signals, overwriting the regular `0` or `1` value of a signal and, thus, injecting a stuck-at fault.

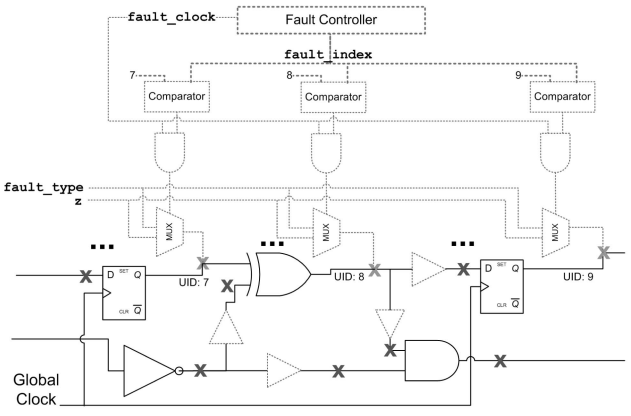

Fig. 3. Fault injection in wires of Gate-Level model (resources for three out of 10 fault injection sites shown).

We point out that additional buffers are used in the diagram of Fig. 3, in order to enable fault injection in the various segments of a wire with fan-out. These buffers enable individual addressing and, thus, fault injection on each of the branches of a fan-out net. Finally, we also note that in order to successfully simulate a Gate-Level module in an RT-Level environment, the delays of all the standard cells are set to zero, matching the default zero propagation delay of an RT-Level model.

3.3 Simulation Flow

After presenting each component of the infrastructure, we now describe how these components are combined to provide the aforementioned capabilities. Fig. 4 presents a flowchart of the procedure, where each of the three distinct components of Fig. 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 are added to the IVM 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 by 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.

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 and 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.

4 Instruction-Level Errors

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 faults in the control logic of a modern microprocessor. Thereby, these ILE types enable us to study the correlation between low-level faults in the hardware implementation of control logic and their instruction-level impact.

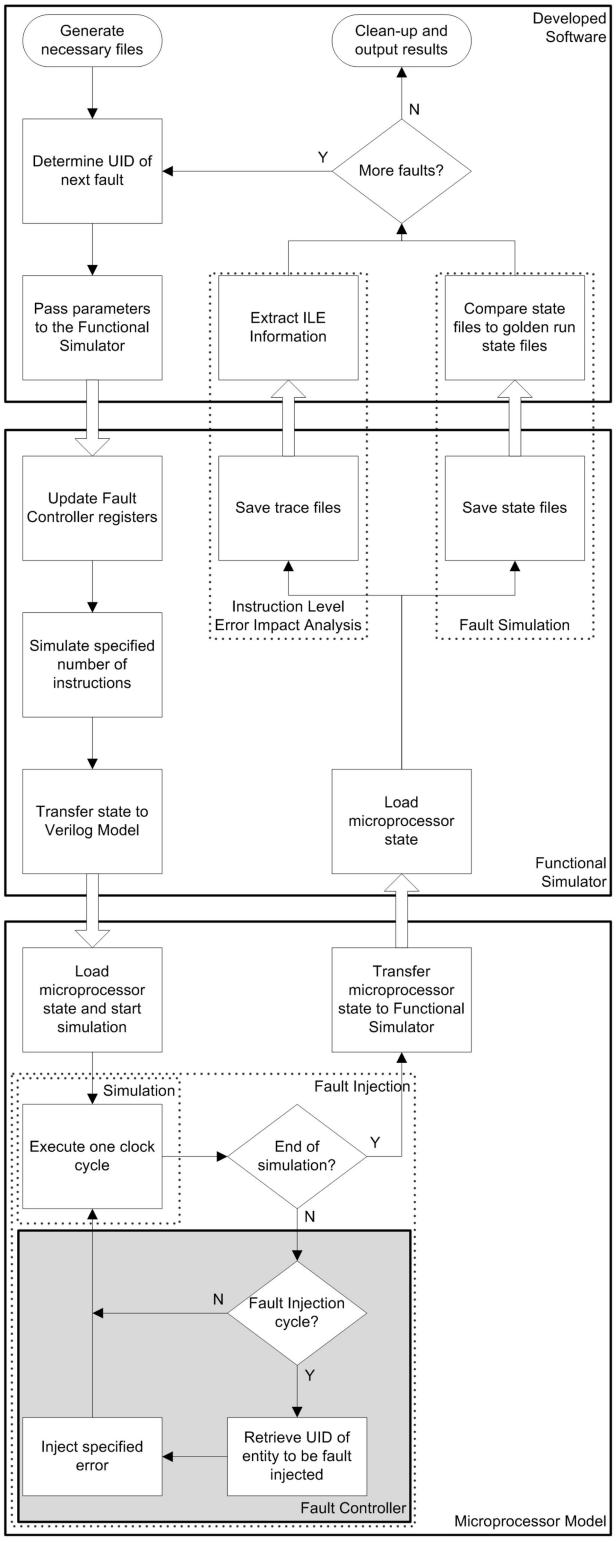

Fig. 4. Infrastructure utilization flowchart.

4.1 ILE Groups & Types

In this study, we consider 13 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 superscalar out-of-order microprocessor, namely

1. the operation that is executed,
2. the operands that are being used,

TABLE 2
Instruction-Level Errors

Group 1: Operation Errors	**Type 1:**	Incorrect operation code used
	Type 2:	Invalid operation code used
Group 2: Operand Errors	**Type 3:**	Incorrect register addressed
	Type 4:	Invalid register addressed
	Type 5:	Premature use of register
	Type 6:	Incorrect immediate operand
Group 3: Execution Errors	**Type 7:**	Incorrect functional unit 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

3. the functional unit where execution takes place,
4. the starting and finishing time of execution, and
5. 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 operation code (op_code) of the instructions executed by the microprocessor, classified as 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 registers and immediate operands through the following ILE types:

- **Type 3:** A register address used by an instruction points to a valid but incorrect register file location.
- **Type 4:** A register address used by an instruction points to an invalid register file location.
- **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

Superscalar 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 3
ILE Classification Information Traced from Various Microprocessor Modules

Module	Information
Decoder	Opcode, Operands, Immediate
Rename	Physical Registers
Execution	Functional Unit Utilization
Scheduler	Issue Time, Replays
Scoreboard	Availability of Operands
Reorder Buffer (ROB)	ROBid, Retirement Cycle/Order

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 execution times 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 than 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, the order in which instructions are scheduled and executed does not necessarily follow the program sequence. 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 Classification of Low-Level Faults as ILE Types

In order to be able to appropriately categorize the impact of a low-level control logic fault 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 once for a golden run, wherein no fault is injected, and subsequently repeated on the fault-injected processor for each fault. 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 from the various modules during each clock cycle is summarized in Table 3. Evidently, the entire microprocessor needs to be simulated in order to accurately assess the impact of a low-level fault on instruction execution. Specifically, the traced information includes:

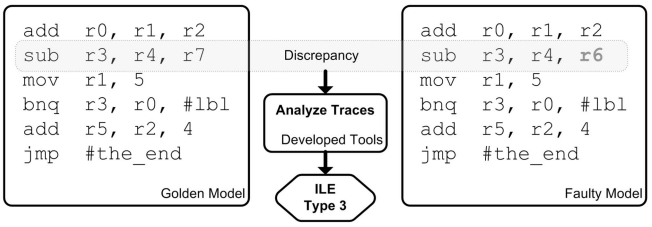

Fig. 5. Example of classifying a low-level fault as a Type 3 ILE (incorrect register used).

1. The op_code of the instruction being executed; this is simply the type of the instruction. Based on this, ILEs of Types 1 and 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 and 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 that 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 and 8 can be identified.
6. The clock cycle at which the instruction starts execution. Based on this, ILEs of Types 9 and 10 can be identified.
7. The clock cycle at which the instruction is expected to finish execution. Based on this, ILEs of Types 11 and 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 supported by the IVM. Based on this, ILEs of Type 13 can be identified.

When fault simulation of a benchmark completes its execution window, automated line-by-line comparison of its trace to the trace of the golden run is performed, as shown in Fig. 5. In the event of a discrepancy, internally developed tools employ certain checks and algorithms to classify the fault to the appropriate ILE type. 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.

5 Experiments

In this section, we demonstrate the error correlation capabilities and the corresponding insight that can be gained through the developed infrastructure. Specifically, we perform a series of simulations wherein low-level faults (either at the RT- or at the Gate-Level) are injected in the control logic of IVM while the latter executes SPEC2000 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 developing cost-effective CED methods.

TABLE 4
Target Module Details

Module	# of entities	# of stdcells	# of stuck-at faults
Scheduler	9,411	170,099	1,159,012
ReOrder Buffer (ROB)	30,735	228,881	1,714,306

5.1 Experimental Setup

Simulation workload. Seven different SPEC2000 benchmarks, namely `bzip2`, `mcf`, `parser`, `vortex`, `gzip`, `gap`, and `cc` are used as the simulation workload. The use of multiple benchmarks ensures variability of the instructions executed through the processor and the control logic that they exercise. Each benchmark is executed by the functional simulator for an initial warm-up period of 50,000 clock cycles, at which point the machine state is transferred to the Verilog model and the execution continues for 2,000 cycles, during which a fault may be injected. On average, 1,297 instructions retire in this window of 2,000 clock cycles.

Target modules. Since our focus is on microprocessor control logic, we target two key control modules: the Scheduler, which controls the allocation of instructions to execution units, and the ROB which controls the order of instruction retirement. Both modules incorporate large buffers to support the number of instructions that can be in-flight, with a combined total of over 40,000 Verilog entities, as shown in Table 4. The Scheduler is relatively small; it contains 32 slots for instructions waiting to be executed and keeps the information needed to identify and correctly issue an instruction. The ROB is much larger because it contains a 64-slot instruction buffer, as well as complementary information about instruction retirement order.

As shown in Table 4, the synthesized Scheduler consists of 170,099 standard cells, while the synthesized ROB consists of 228,881 standard cells. Even though the ROB has over 20K (228 percent) more storage elements than the Scheduler, it only uses 34 percent more standard cells. This is explained since, despite the fact that the ROB uses much larger buffers, the control logic involved is rather small and is only limited to the proper retirement of instructions. On the other hand, the Scheduler performs complicated tasks such as checking whether operands are ready, whether an instruction can be issued avoiding structural hazards, etc., which involve much more control logic. The two modules employed in this study are quite diverse, one being a control-logic heavy and the other being buffer-heavy. Hence, we expect the results of our analysis to carry over to other modules that are mainly concerned with instruction execution flow.

Injected faults. Both stuck-at and transient faults are considered in this study. For the transient fault model, we

inject a bit-flip in every simulated clock cycle, which translates to 2,000 fault simulation runs per fault location.

While simulating the RT-Level versions of the two modules, all entities (presented in Table 4) are injected. At the gate-level, however, the number of faults is very large, since it includes faults both in the flip-flops and in the combinational logic, so we resort to sampling, with a sample size of $c = 30\%$. For the Scheduler, where the total number of faults is $N_{Scheduler} = 1{,}159{,}012$, this translates to a sample size of $n_{Scheduler} = 347{,}703$ faults, while for the ROB, where $N_{ROB} = 1{,}714{,}306$ faults, this translates to a sample size of $n_{ROB} = 514{,}290$ faults. To assess the error incurred due to sampling, we use the following equation, defined in [14]:

$$C_{0.99} = c \pm \epsilon, \epsilon = \frac{a^2 k}{2N_i}\sqrt{1 + \frac{4N_i c(1-c)}{a^2 k}}, \quad (1)$$

where ϵ is the incurred error, $C_{0.99}$ is the range of fault coverage within which the true coverage lies with a confidence interval of 99 percent, N_i is the total number of faults, c is the fraction of faults to be simulated, $a = 2.60$ to achieve a confidence interval of 99 percent, and $k = 1$ since the total population is large. Thus, our sample size of $c = 30\%$ yields an error of $\epsilon = 0.01\%$, which is more than adequate for the purpose of our analysis.

Computational power. The experiments are performed on a Quad-core Xeon 3.33 GHz with 16 GB of memory.

5.2 Results and Analysis

In this section, we present the results of evaluating the impact of low-level faults on the instruction level. The results are divided in three sections:

1. First, we present results which reveal the correlation between RT-Level faults and the instruction-level errors that they cause.
2. Second, we present a comparative analysis of the impact of RT- vis-a-vis Gate-Level faults on instruction-level execution. These results corroborate that the above correlation remains valid independent of the level at which fault injection is performed.
3. Third, we present a comparative analysis of the impact of stuck-at faults vis-a-vis transient errors. Once again, the results corroborate that the correlation between low-level faults and instruction-level errors holds true, independent of the injected fault type.

5.2.1 Instruction-Level Impact of RT-Level Stuck-At Faults

Fault simulation statistics. As a first set of results, we present cumulative data regarding the fault simulations performed. Specifically, Table 5 reports the percentage of the 80,804 injected faults (for both the Scheduler and the ROB) 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 16 and 42 percent. 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. 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). This high percentage of application-level masking highlights the advantage of using workload information to develop efficient CED techniques.
- Among the faults resulting in an ILE, an average of 99.3 percent also cause an architectural error. This high percentage elucidates the fact that the proposed ILE types constitute an effective way of capturing incorrect workload execution. The few faults that cause an ILE but do not affect the architectural state area attributed to either architectural masking (e.g., a fault that corrupts an instruction which never commits due to speculative execution) or performance faults (e.g., a fault that delays the availability of a functional unit and causes the workload to take longer to execute) [15]. We also note that, in our experiments, none of the faults may result in an incorrect architectural state but not cause an ILE.
- 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 5, the average number of simultaneously activated ILE types is only 1.24; 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 remaining unaffected. Thus, early detection of such ILEs can guide simple local operations toward restoring the correct state of the microprocessor.

TABLE 5
Results on SPEC2000 Integer Benchmarks

SPEC Benchmark	Faults Resulting in ILEs[1]	Average # of ILEs Activated Simultaneously
bzip2	39.3% (98.6%)	1.32
mcf	16.6% (98.2%)	1.19
parser	42.0% (99.9%)	1.39
vortex	39.2% (99.8%)	1.10
gzip	17.4% (99.4%)	1.26
gap	17.0% (99.9%)	1.14
cc	38.2% (99.5%)	1.31
Average	**29.9% (99.3%)**	**1.24**

[1] The percentage shown in the parenthesis is the percentage of faults resulting in an ILE that also corrupt the architectural state of the microprocessor in the injection period.

Benchmark consistency. 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 error correlation information may be

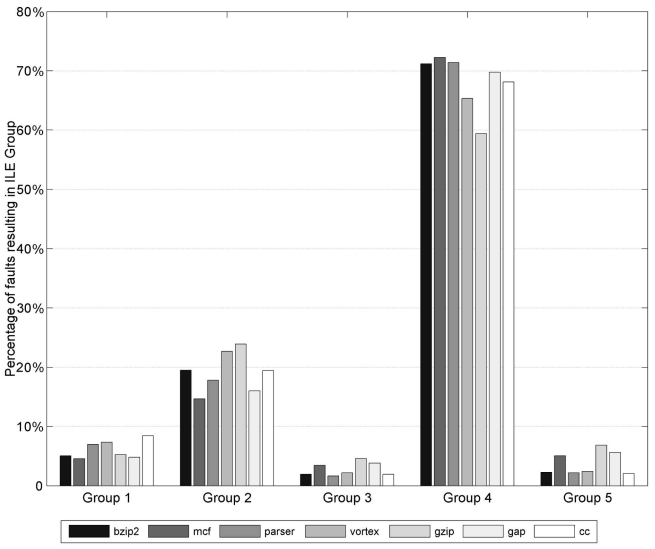

Fig. 6. Percentage of stuck-at faults causing each ILE group for each of the seven benchmarks.

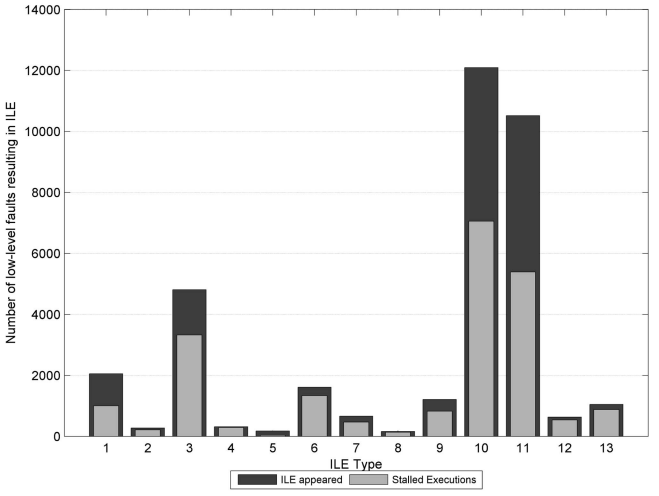

Fig. 7. Average number of stuck-at faults causing each ILE type and subsets causing stalled execution.

questioned. Therefore, in Fig. 6, 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. These observations corroborate that the obtained error 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 (60-75 percent) result in timing errors in all benchmarks, implying that, independent of the workload utilized, 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 [16].

ILE distribution. The third set of presented results relates to the occurrence frequency of each of the 13 ILE types described in Section 4.1. The average number of stuck-at faults resulting in each ILE type over the seven benchmarks is presented in Fig. 7. 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. 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 one to three 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 and 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.
- Among the faults resulting in an ILE most (up to 80 percent) lead to a pipeline stall. In other words, while the initial error may only manifest in one aspect of instruction execution, if the error is not corrected promptly it will most likely have an avalanche effect that will eventually stall the processor. Moreover, our intuition is that the reported percentage is an underestimate due to the limited fault simulation window of 2,000 cycles.

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 specific ILE types.

ILE appearance time. 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 microprocessor. This

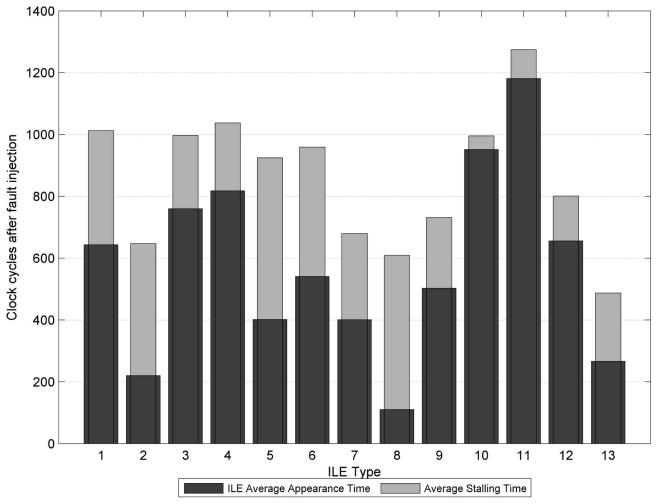

Fig. 8. Average time stamp of ILE identification and subsequent pipeline stalling (in clock cycles).

information is provided in Fig. 8. While the activation time of an ILE may depend on the actual sequence of executed instructions, averaging the results over the seven SPEC2000 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 types 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 Fig. 7), 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 and 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 and 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 Type 7, 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. Thus, they can be used to fine-tune error tolerance methods that employ checkpoints to examine and restore the microprocessor state [10]. 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, Fig. 8 shows that the activated ILEs appear no later than 1,200 cycles after fault injection; this ensures that the window of 2,000 clock cycles that we observe is sufficiently long for the performed analysis. While ILEs that would require a much larger simulation window to appear may exist, their number is relatively small and would not alter the profile of the results.

5.2.2 Impact Comparison of RT- versus Gate-Level Faults

Impact consistency. The first set of results compares the accuracy of assessing the impact of low-level faults on instruction execution at the RT- versus the Gate-Level. Specifically, Figs. 9 and 10 present the classification of RT- and Gate-Level faults into the ILE types that they cause, for each of the seven benchmarks. The results are presented separately for the Scheduler and the ROB since the Scheduler is a control logic-heavy module whereas the ROB is a buffer-heavy module. Based on the results, we can make the following observations:

- The most important observation is a pronounced consistency in the types and frequency of ILEs caused by RT- versus Gate-Level faults. Indeed, for each benchmark, the distribution of RT-Level faults to the 13 ILE types is strongly correlated with the distribution of Gate-Level faults to the same, as presented in Table 6. On average, the correlation coefficient[2] is 93 percent for both the Scheduler and the ROB. This critical observation corroborates that RT-Level fault analysis provides sufficiently accurate results with respect to its Gate-Level counterpart.
- The next observation is that the distributions are consistent across the various benchmarks, confirming the results presented in the previous section. Hence, it is evident that the type of ILE caused by a low-level fault is, typically, independent of the instruction subset that is utilized.
- A final observation is that some ILE types appear never to be caused by Gate-Level faults. These ILE types have already a very small frequency of occurrence due to RT-Level faults, which is further diminished during Gate-Level fault simulation because of sampling.

Fault simulation speed. The next set of results investigates how the simulation speed is affected by the use of Gate-Level modules. Table 7 compares the average simulation

2. If R is the vector containing the 13 values representing the percentage of the faults that result in each ILE Type for the RT-Level, and G is the same vector for the Gate-Level ($\bar{R}$ and $\bar{G}$ being the respective average values), then the correlation coefficient c is calculated as

$$c = \frac{\sum_{n=1}^{13}(R_n - \bar{R})(G_n - \bar{G})}{\sqrt{\left(\sum_{n=1}^{13}(R_n - \bar{R})^2\right)\left(\sum_{n=1}^{13}(G_n - \bar{G})^2\right)}}.$$

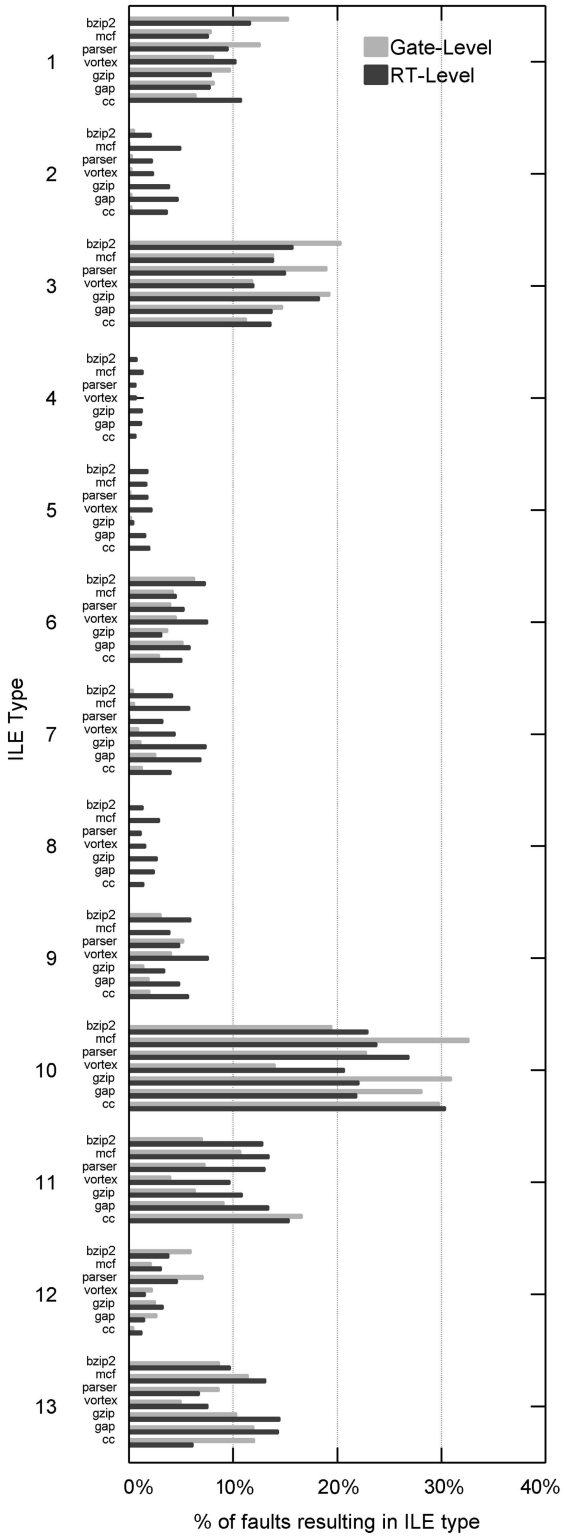

Fig. 9. Comparison between ILE types caused by RT- versus Gate-Level faults in the Scheduler.

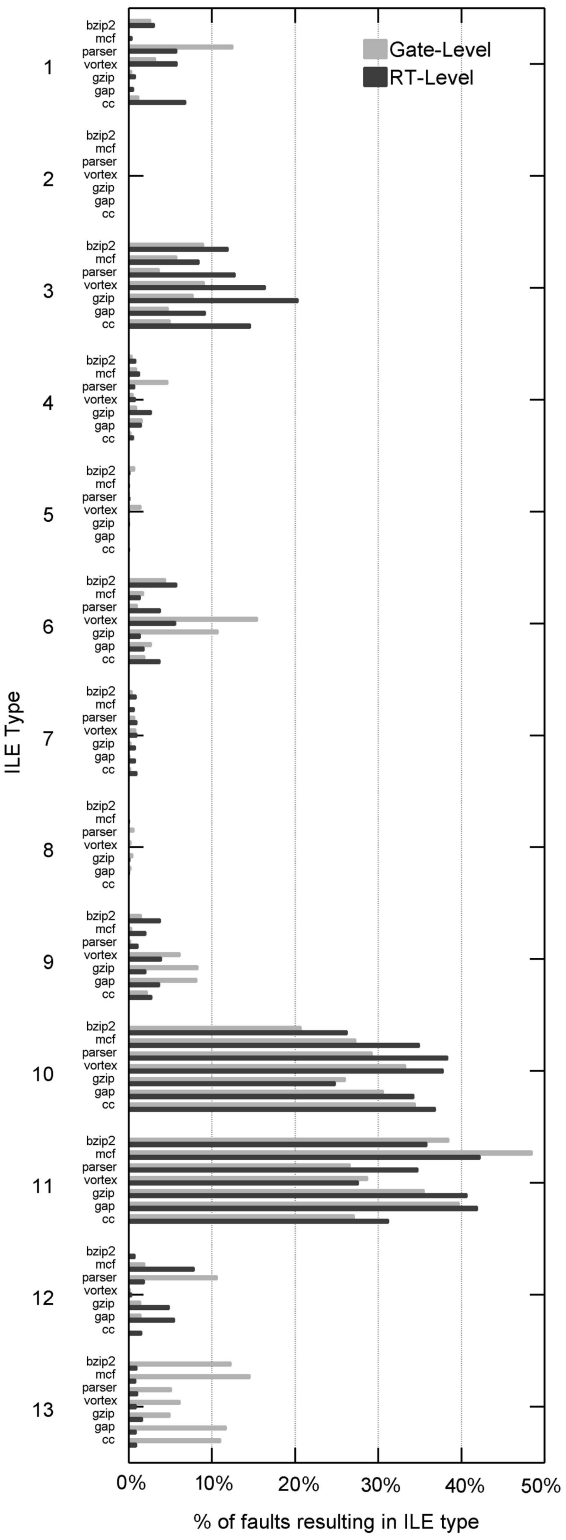

Fig. 10. Comparison between ILE types caused by RT- versus Gate-Level faults in the ROB.

time per fault for the various configurations. The first row provides the baseline, where all modules are simulated at the RT-Level, while the following rows indicate the overhead when one or both of the target modules are simulated at the Gate-Level. As may be observed, this makes the simulation over an order of magnitude slower. Taking into account that, even with 30 percent fault sampling, using the Gate-Level Scheduler and ROB requires, on average, $5.8\times$ ($5.0\times$ for Scheduler, $6.6\times$ for ROB) more fault simulations than their RT-Level counterparts, a similar fault analysis at the Gate-Level is $260\times$ more expensive, rendering it almost infeasible for extensive studies. Therefore, performing the simulations at the RT-Level is strongly desired, especially since the accuracy of the obtained results is up to par, as confirmed by the above results.

TABLE 6
Results on SPEC2000 Integer Benchmarks

SPEC Benchmark Name	Correlation Coefficient for Scheduler[1]	Correlation Coefficient for ROB[1]
bzip2	91.66%	94.07%
mcf	96.29%	93.32%
parser	92.47%	91.63%
vortex	93.96%	93.84%
gzip	93.02%	90.96%
gap	94.65%	95.74%
cc	95.60%	93.46%
Average	**93.95%**	**93.29%**

5.2.3 Impact Comparison of Stuck-at versus Transient Faults

Transient fault classification. We now compare the impact of two different fault models, namely stuck-at faults and bit-flip transients, on instruction execution. We note that, while classifying the impact of a stuck-at fault to an ILE is rather straightforward, doing so for a transient fault is more complicated. Indeed, transients in the same location may result in a different ILE type depending on the time of fault injection during the simulation. Therefore, for each fault we perform 2,000 simulations, each time injecting it at a different clock cycle and we collect the probability with which this fault will lead to an ILE of each type. For example, assume that a stuck-at fault in a register causes an ILE of Type 3, while a bit-flip in the same register results in an ILE of Type 3 if injected in clock cycles 10, 500, or 1,200 and as ILE of Type 11 if injected in clock cycles 250 or 1,500. Then, this transient fault contributes 0.6 to the number of transient faults that result in an ILE of Type 3 and 0.4 to the number of transient faults that result in an ILE of Type 11. In contrast, the corresponding stuck-at fault contributes 1 to the number of faults resulting in an ILE of Type 3.

Impact consistency. Fig. 11 shows the combined results for the Scheduler and the ROB for the seven different benchmarks. The results reveal a very high consistency between the distribution of stuck-at and transient faults to the corresponding ILEs (i.e., an average correlation coefficient of 98 percent). This consistency can be further explained by examining the impact of individual transient faults. While a transient may result in various ILE types depending on its time of injection, it turns out that, most of the time (over 80 percent on average), it causes the same ILE. Hence, even if one is interested in developing CED methods for transient bit-flips, assessing their instruction-level impact through stuck-at fault simulations would provide sufficiently accurate results. Moreover, as explained in the previous paragraph, assessing instruction-level impact through simulation of stuck-at faults is far less time consuming than through transient faults, since only one simulation is necessary per fault location.

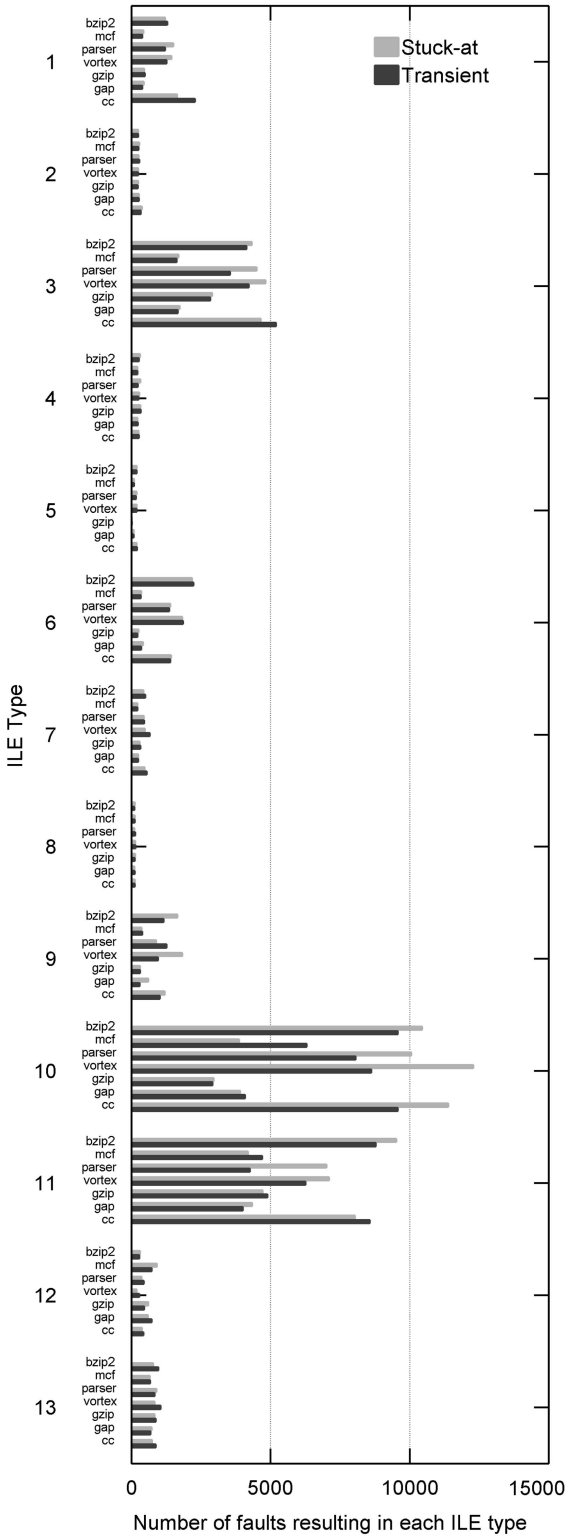

Fig. 11. Comparison between ILE types caused by stuck-at versus transient faults in the Scheduler and the ROB.

TABLE 7
Fault Simulation Speed Comparison

Fault Simulated Modules	Seconds per Fault Simulation	Time Overhead (Comp. to RTL)	Number of Simulated Faults[1]	Overall Overhead (Comp. to RTL)
All RTL modules	3.26	-	80,292	-
Gate-level Scheduler	35.04	10.7x	407,173	54.5x
Gate-level ROB	42.20	12.9x	533,112	85.9x
Gate-level Sch., ROB	79.02	24.2x	861,993	260.2x

[1] The Gate-Level fault list is a 30% uniform sample

6 Conclusions

In order to develop cost-effective CED methods which leverage the ability of modern microprocessors to suppress a high percentage of errors at the application level, a thorough understanding of the impact of low-level faults on program execution is necessary. To this end, the infrastructure reported herein, which we developed around an Alpha-like high-performance microprocessor, enables injection, simulation, and classification of low-level faults into instruction-level error types. Extensive experimentation with this infrastructure provided great insight regarding the relative importance of low-level faults, as gauged by the activation frequency and latency of the instruction-level error types that they cause. Furthermore, it revealed a profound instruction-level impact consistency between RT- and Gate-Level faults, as well as between stuck-at and transient faults. Besides CED, the capabilities of the developed infrastructure may be utilized to provide similar insight and guidance for various other design robustness endeavors.

Acknowledgements

This work is supported by a generous gift from Intel Corp. The first two authors contributed equally to this work. The second author performed this research while being a visiting student at Yale University. Preliminary versions of parts of the results reported herein were presented at the 2008 International Test Conference [17] and the 2009 VLSI Test Symposium [18]. The authors would like to thank Professor 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," *Proc. IEEE Int'l Test Conf.,* pp. 524-533, 1998.

[3] S. Mitra and E.J. McCluskey, "Which Concurrent Error Detection Scheme to Choose?," *Proc. IEEE Int'l Test Conf.,* pp. 985-994, 2000.

[4] K. Mohanram and N.A. Touba, "Cost-Effective Approach for Reducing Soft Error Rate in Logic Circuits," *Proc. IEEE Int'l Test Conf.,* pp. 893-901, 2003.

[5] S. Almukhaizim, P. Drineas, and Y. Makris, "Entropy-Driven Parity-Tree Selection for Low-Overhead Concurrent Error Detection in Finite State Machines," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* vol. 25, no. 8, pp. 1547-1554, Aug. 2006.

[6] J.C. Lo, "A Hyper Optimal Encoding Scheme for Self-Checking Circuits," *IEEE Trans. Computers,* vol. 45, no. 9, pp. 1022-1030, Sept. 1996.

[7] C. Metra, D. Rossi, M. Omana, A. Jas, and R. Galivanche, "Function Inherent Code Checking: A New Low Cost On-Line Testing Approach For High Performance Microprocessor Control Logic," *Proc. European Test Symp.,* pp. 171-176, 2008.

[8] N.J. Wang, A. Mahesri, and S.J. Patel, "Examining ACE Analysis Reliability Estimates Using Fault Injection," *ACM SIGARCH Computer Architecture News,* vol. 35, no. 2, pp. 460-469, 2007.

[9] D. Burger and T.M. Austin, "The SimpleScalar Tool Set," Technical Report CS-TR-97-1342, Version 2.0., Univ. of Wisconsin, Madison, 1997.

[10] N.J. Wang and S.J. Patel, "Restore: Symptom Based Soft Error Detection in Microprocessors," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 30-39, 2005.

[11] N.J. Wang, J. Quek, T.M. Rafacz, and S.J. Patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 61-70, 2004.

[12] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: the MEFISTO Tool," *Proc. Int'l Symp. Fault-Tolerant Computing,* pp. 66-75, 1994.

[13] 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," *IEEE Trans. Very Large Scale Integration (VLSI) Systems,* vol. 16, no. 6, pp. 693-706, June 2008.

[14] V.D. Agrawal and H. Kato, "Fault Sampling Revisited," *IEEE Design and Test of Computers,* vol. 7, no. 4, pp. 32-35, Aug. 1990.

[15] N. Karimi, M. Maniatakos, C. Tirumurti, A. Jas, and Y. Makris, "Impact Analysis of Performance Faults in Modern Microprocessors," *Proc. IEEE Int'l Conf. Computer Design,* pp. 91-96, 2009.

[16] M. Maniatakos, N. Karimi, Y. Makris, A. Jas, and C. Tirumurti, "Design and Evaluation of a Timestamp-Based Concurrent Error Detection Method (CED) in a Modern Microprocessor Controller," *Proc. IEEE Int'l Symp. Defect and Fault Tolerance of Very Large Scale Integration Systems,* pp. 454-462, 2008.

[17] N. Karimi, M. Maniatakos, Y. Makris, and A. Jas, "On the Correlation Between Controller Faults and Instruction-Level Errors in Modern Microprocessors," *Proc. IEEE Int'l Test Conf.,* pp. 24.1.1-24.1.10, 2008.

[18] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and Y. Makris, "Instruction-Level Impact Comparison of RT- versus Gate-Level Faults in a Modern Microprocessor Controller," *Proc. IEEE Very Large Scale Integration Test Symp.,* pp. 9-14, 2009.

Michail Maniatakos received the BS and MS degrees in computer science and embedded systems from the University of Pireaus, Greece, in 2006 and 2007, respectively, as well as the MS degree in electrical engineering from Yale University, New Haven, CT, in 2008, where he is currently a PhD candidate. His current research interests include test and reliability of modern microprocessors and computer architecture. He is a student member of the IEEE.

Naghmeh Karimi received the BS, MS, and PhD degrees in computer engineering from the University of Tehran, Iran, in 1997, 2002 and 2009, respectively. Her masters thesis was on testability enhancement at the register transfer level and the PhD thesis was on concurrent error testing and reliability enhancement. Between 2007 and 2009, she was a visiting researcher at Yale University. She is currently a post-doctoral researcher at Duke University. Her research interests include design for testability, concurrent testing, fault tolerance, and reliability enhancement. She is a student member of the IEEE.

Chandrasekharan (Chandra) Tirumurti is a research scientist with the Validation and Test Solutions group at Intel Corporation based in Santa Clara, CA. His current focus is on strategic manufacturing test initiatives for mainstream CPUs. An alumnus of Indian Institute of Technology, Kharagpur, India, has wide experience in many areas of CAD and design, including simulation, data path synthesis, defect oriented testing, and fault tolerance. He has published several papers in the areas of test and fault tolerance. He mentors funded research and SRC projects actively for Intel and is an avid cricketer. He is a member of the IEEE.

Abhijit Jas received the BE degree in computer science and engineering from Jadavpur University, Kolkata, India, in 1996. He secured the first rank among all graduating students from the college of engineering. He received the MS and PhD degrees in electrical and computer engineering from the University of Texas at Austin, in 1999 and 2001, respectively. He is a component design engineer with the Validation and Test Solutions group at Intel Corporation in Austin, TX. His current focus is on scalable and modular test access mechanism architecture for system-on-a-chip products. He has published several papers in leading conferences and journals in the areas of VLSI testing and fault tolerance. He mentors several academic research projects funded by Intel. He was a corecipient of the 2001 Best Paper Award at the VLSI Test Symposium. He serves on the technical program committee of several IEEE conferences and workshops. He was the program chair of the International Test Synthesis Workshop in 2009. He is a member of the IEEE.

Yiorgos Makris received the diploma of computer engineering and informatics from the University of Patras, Greece, in 1995, and the MS and PhD degrees in computer science and engineering from the University of California, San Diego, in 1997 and 2001, respectively. He, then, spent over 10 years as a faculty of Electrical Engineering and of Computer Science at Yale University, and he is currently an associate professor of Electrical Engineering at The University of Texas at Dallas, where he leads the Trusted and Reliable Architectures (TRELA) Research Group. His current research interests include soft-error mitigation in digital circuits, machine learning-based testing of analog/RF circuits, mitigation of hardware Trojans, as well as test and reliability of asynchronous circuits. He serves on the organizing and program committees of many conferences in the areas of test and reliability and is the program chair for the 2011 Test Technology Education Program (TTEP) of the IEEE Test Technology Technical Council (TTTC). He is a senior member of the IEEE.