Workload-Cognizant Concurrent Error Detection in the Scheduler of a Modern Microprocessor

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

Abstract—We present a Concurrent Error Detection (CED) scheme for the Scheduler of a modern microprocessor. The proposed CED scheme is based on monitoring a set of *invariances* imposed through added hardware, violation of which signifies the occurrence of an error. The novelty of our solution stems from the *workload-cognizant* way in which these invariances are selected so that they leverage the application-level error masking inherent in program execution. Specifically, in order to ensure cost-effectiveness of the hardware employed to construct these invariances, we make use of information regarding the type and frequency of errors affecting the typical workload of the microprocessor. Thereby, we identify the most susceptible aspects of instruction execution and we accordingly distribute CED resources to protect them. Our approach is demonstrated on the Scheduler of an Alpha-like superscalar microprocessor with dynamic scheduling, hybrid branch prediction and out-of-order execution capabilities. Using an extensive fault-simulation infrastructure that we developed around this microprocessor, we profile the impact of Scheduler faults across a variety of different SPEC2000 benchmarks. Based on the results, we construct a CED scheme which monitors the time and location of instruction execution, the executed operation, the utilized resources, as well as the executed and retired sequence of instructions. At a hardware cost of only 32 percent of the Scheduler, the corresponding CED scheme detects over 85 percent of its faults that affect the architectural state of the microprocessor. Furthermore, over 99.5 percent of these faults are detected before they corrupt the architectural state, while the average detection latency for the remaining faults is in the order of a few clock cycles, implying that efficient recovery methods can be developed.

Index Terms—Concurrent error detection, microprocessor, scheduler, invariance.

1 Introduction

The physical challenges incurred by the rapidly shrinking feature size and reduced power supply voltage of deep submicron semiconductor fabrication technologies continue to give rise to various design robustness concerns. While soft errors occurring due to strikes by neutrons or alpha particles, which may lead to corresponding single event upsets (SEUs) in memory bits, or single event transients (SETs) in combinational logic have received the lion's share of attention, they only constitute part of the problem. Indeed, various other faults related to crosstalk and ground-bounces, design marginalities, process variations and corner operating conditions [1], [2], [3], [4], [5] are starting to cause errors and to play an increasingly important role. Ranging in duration from single events to permanent faults, such errors have revived interest in concurrent error detection (CED) and/or correction methods that may ameliorate or resolve their effect [6], [7].

The pluralism of CED methods that have been proposed in the past implies that no one solution fits the needs of every circuit. Furthermore, it stresses the fact that applying generic solutions across the board for an entire design typically incurs prohibitive cost, often times unnecessarily and without providing commensurate coverage. Indeed, in order to maintain cost-effectiveness, it is important to understand the specific vulnerabilities and requirements of a circuit and accordingly tailor the CED solution.

Modern microprocessors, for example, exhibit a high degree of application-level error masking. In other words, many of the errors that occur due to the aforementioned reasons are suppressed or have a low probability of affecting the programs that are typically executed by the microprocessor. Indeed, the multitude of functional units and stages in deeply pipelined superscalar microprocessors, along with advanced architectural features such as dynamic scheduling and speculative execution, imply that rather complex conditions need to be satisfied for an error to affect the architectural state of the microprocessor. Furthermore, it is well known that the probability with which faults are suppressed is asymmetric [8], implying that not all faults

- *N. Karimi is with the Department of Electrical Engineering, Duke University, Durham, NC 27708. E-mail: naghmeh.karimi@duke.edu.*
- *M. Maniatakos is with the Department of Electrical Engineering, Yale University, 10 Hillhouse Avenue, New Haven, CT 06520-8267. E-mail: michail.maniatakos@yale.edu.*
- *A. Jas is with the Design and Technology Solutions Group, Intel Corporation, Austin, TX 78746. E-mail: abhijit.jas@intel.com.*
- *C. Tirumurti is with the Design and Technology Solutions Group, Intel Corporation, Santa Clara, CA 95050. E-mail: chandra.tirumurti@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 14 June 2009; revised 15 Mar. 2010; accepted 30 June 2010; published online 6 Dec. 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-2009-06-0280.
Digital Object Identifier no. 10.1109/TC.2010.265.

are equally critical. Hence, high-level methods that aim to globally monitor the most susceptible aspects of instruction execution, rather than to locally check the result of every fine-grain hardware entity, appear to be the most promising direction towards developing cost-effective CED solutions.

To this end, in this paper we develop a workload-cognizant CED scheme for the Scheduler[1] of a modern microprocessor, based on detailed information regarding the impact of faults on the instruction execution of typical programs. Specifically, we introduce a model of Instruction-Level Errors (ILEs) and we employ an extensive fault simulation infrastructure that we have developed around a modern microprocessor model in order to understand the relative importance of the various ILEs that are caused by faults in the Scheduler of the microprocessor. Guided by the results of this analysis, we incorporate additional hardware that monitors the most vulnerable aspects of instruction execution. Specifically, we predict and validate the time and the functional unit where an instruction is executed, along with four hardware-imposed *invariances* (i.e., properties which hold true during error-free operation but which are violated in the presence of errors), which validate the operation code and the operands of an instruction, as well as the sequence in which instructions are executed and retired. Extensive fault simulation-based evaluation validates that workload information can, indeed, drive the development of cost-effective CED methods.

The remainder of this paper is organized as follows: In Section 2, we discuss related work in CED. Then, in Section 3, we introduce the microprocessor model that is used to demonstrate our CED scheme, the fault simulation infrastructure that has been previously developed around it and its limitations, as well as the enhancements that we made for the purpose of our work. In Section 4, we provide more details about the Scheduler of this microprocessor, which is the target of our CED scheme. Next, in Section 5, we use the aforementioned infrastructure to analyze the impact of Scheduler faults on typical workload executed on the microprocessor and we draw key observations that drive the development of our CED scheme, the details of which are given in Section 6. Finally, extensive fault simulation results demonstrating the effectiveness of the proposed CED scheme are presented and discussed in Section 7 and conclusions are drawn in Section 8.

2 Related Work

Various CED solutions [6], [7] have been proposed in the past to detect faults or errors occurring during normal operation of a circuit after it is deployed in the field. Among those, the simplest approach is duplication, wherein a replica of the circuit is added to the design, possibly diversely implemented (e.g., through the use of dual-rail logic [9]) to avoid common mode failures [10]. The original and the replica serve as predictors of the functionality of each other and a simple comparator indicates any discrepancy in their outputs, thus detecting potential malfunctions. While simple, this technique is prohibitively expensive. To reduce the overhead imposed by duplication-based techniques, partial duplication has been suggested to detect the faults in the most critical parts of the design [8].

Another very popular CED approach has been the use of various codes, especially within the context of finite state machine (FSM) controllers. Several redesign and resynthesis methods are described in [11], [12], wherein parity or various unordered codes are employed to encode the states of the circuit. Utilization of multiple parity bits is also examined in [13] within the context of FSMs. These methods guarantee latency-free error detection; on the down side, they are intrusive and expensive. Nonintrusive CED methods have also been proposed. Implementations based on Hamming, Bose-Lin, and Berger codes are presented in [14], [15], and [16], respectively, while parity-based CED methods are described in [6], [17], [18]. While the aforementioned methods guarantee latency-free detection of all errors, their cost is often prohibitive. Trading off the incurred cost by allowing a nonzero, yet bounded error detection latency has also been investigated [19].

At a coarser level, an attempt to identify inherent invariance either at the gate-level [20] or at the RTL [21] of a design has been made. Such invariance can be monitored during the normal operation of a circuit to identify errors that cause it to be violated. In [20] such invariance is mined from the gate-level of a controller implementation in the form of assertions, which are evaluated through simulation in order to select a cost-effective appropriate subset. The same principle governs the approach in [21]; therein, however, invariance is identified through a path-construction algorithm, which exploits inherent transparency channels that exist in the RTL description of a modular design. More recently, the method proposed in [22] also leverages inherent invariant codes to perform CED in the decoder of the same microprocessor that we employ in this study.

At an even higher architectural level, several concurrent error detection, and/or correction methods have been proposed. The concept of watchdog processors, which compute control-flow signatures and compare them to expected correct values, known at compilation time, is proposed in [23], [24]. Concepts akin to instruction-level duplication and comparison to identify erroneous results are examined in [25], [26]. In [27], the authors examine the vulnerability of different parts of a microprocessor to soft errors and recommend various strategies (including register file protection with codes, parity coding to protect instruction words, and a timeout counter to flush the pipeline when no activity occurs for prolonged periods) to detect/correct such errors. Similar analysis is performed in [28], based on the concept of Architectural Vulnerability Factor (AVF), which prioritizes microprocessor modules based on their susceptibility. Such metrics can prove very useful in guiding allocation of CED resources.

3 Study Framework

This research builds upon a previously developed fault simulation infrastructure, which is presented in detail in [29]

1. While CED methods involving encoding through Residue Number System (RNS) or similar codes have effectively been applied to the data paths of modern microprocessors, the same is not true for their controllers. Hence, our work focuses on control modules that due to their complex structure are still left largely unprotected.

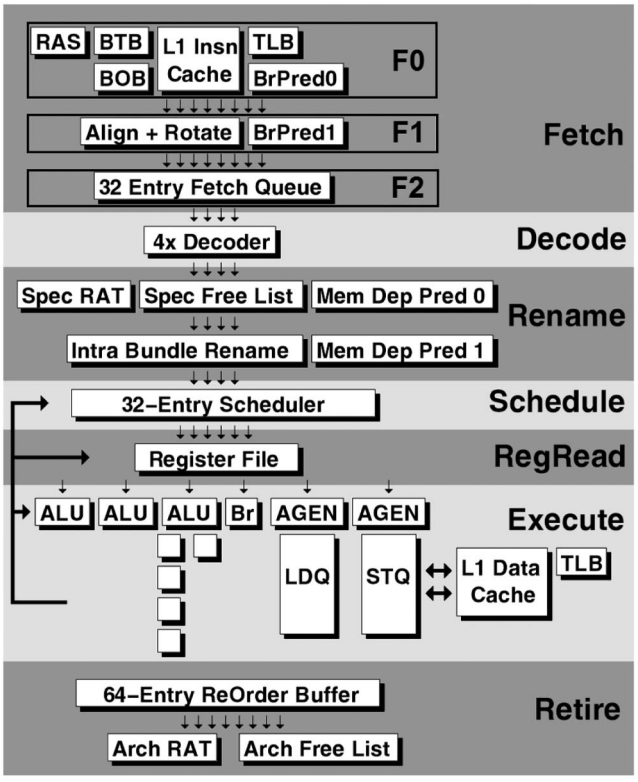

Fig. 1. Block diagram of IVM [27].

and which we summarize herein for the purpose of completeness. We start by briefly introducing the microprocessor that we will use as the test vehicle in our investigation along with the capabilities and limitations of the simulation infrastructure that has been previously developed around it by other researchers. Then, we discuss the fault simulation enhancements that we added to this infrastructure to support the development of the proposed workload-cognizant CED method.

3.1 Microprocessor Model and Limitations

The Microprocessor model used in this study is the Verilog implementation of an Alpha-like microprocessor, called IVM (Illinois Verilog Model) [27], [30]. 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 six functional units. Fig. 1 shows the block diagram of IVM, as presented in [27].

The complexity of IVM reflects most of the features of modern, high-performance microprocessors. Furthermore, it allows simulation of the execution of actual workload, such as the SPEC2000 benchmarks. Thus, it enables a realistic investigation of the efficiency of CED schemes applied to modern microprocessors. Along with the Verilog implementation of IVM, we also make use of a functional simulator, which is a part of the Simplescalar tool suite and supports the full instruction set of the Alpha 21264 microprocessor [31]. This capability is crucial because it enables us to circumvent the limitations of IVM, which does not support system calls and floating point instructions. Such cases are handled by transferring the simulation state to the functional simulator, executing the corresponding instructions, and transferring the new state back to the Verilog model to resume simulation.

Another limitation of IVM is that due to certain coding techniques used at the RT-Level model, it is not synthesizable; hence gate-Level fault simulation cannot be performed. In fact, to perform fault simulation in IVM, an approach of stopping the simulation, altering the state of the microprocessor, and then resuming the simulation is employed. Although this fault injection approach is effective when studying the impact of single-cycle transient errors, such as those caused by alpha particle strikes, it is extremely inefficient for other fault models, such as stuck-at faults or transient errors of longer duration caused by operational marginalities.

3.2 Enhanced Simulation Capabilities

In order to alleviate the aforementioned limitation, we enhanced the IVM infrastructure to support efficient injection and simulation of both stuck-at and transient faults of arbitrary starting time and duration, while executing SPEC2000 benchmarks. This is achieved by employing an RT-Level fault simulator, which we developed and presented in detail in [29], wherein fault injection is performed using a method similar to the parallel saboteurs technique described in [32]. Specifically, the Verilog model of each target module is mutated and a Fault Controller module is added to control all fault injection parameters, including the location, type, as well as the start and stop injection times for each fault. In this method, a unique identification number, called UID, is given to each entity (i.e., register or wire) of the fault simulation target module. Then during simulation, the Fault Controller is responsible for fault injection. In each clock cycle, one bit of one entity is accessed and set to the faulty value. When the Fault Controller activates the fault clock (i.e., the signal that controls the fault simulation starting and stopping clock cycle), each module compares the broadcasted UID (i.e., the UID of the target fault simulation signal which is set by the Fault Controller) to the UIDs of its internal entities. If a match is found, the module modifies the corresponding bit, as specified by the Fault Type coming from the Fault Controller to the module.

Fig. 2 shows a high-level diagram of this method, which allows injection of either stuck-at or transient faults with user-defined activation times to any entity defined in the RT-Level Verilog model (dotted lines indicate hardware added for fault simulation). As shown in this figure, each storage element or wire is driven by a multiplexer which is controlled by the Fault Controller to inject the appropriate value to the intended location during the active fault injection window.

In addition to the enhanced fault injection and simulation capabilities, we have also incorporated extensive tools to the IVM infrastructure for collecting, processing and reporting fault simulation results. Furthermore, the developed tools can be used to save any trace or state files

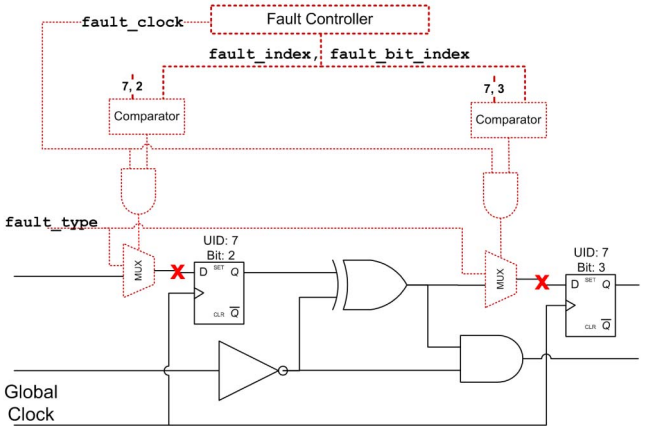

Fig. 2. RT-Level fault injection method.

requested and perform comparisons between golden (fault-free) and faulty model executions. 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. This collection of tools proves invaluable in assessing the impact of faults on the typical programs executed by the microprocessor and, thereby, developing the workload-cognizant CED method proposed herein.

4 Scheduler Module

In this section, we describe in more detail the module targeted by our workload-cognizant CED method, namely the Scheduler, which is one of the key control modules in any modern microprocessor with advanced architectural features. In IVM, the Scheduler is a dynamic module containing an array of up to 32 instructions waiting to be issued, from which up to six instructions are issued in each clock cycle. Each instruction coming to the Scheduler resides in this buffer until an acknowledgment is received from the execution unit that it can start execution. At this time, the corresponding location in the scheduler waiting-list is cleared for use by another newly arriving instruction. The Scheduler issues instructions out-of-order after considering the availability of instructions of various types in the buffer, as well as the existence of structural or data hazards.

During instruction execution, avoidance of structural and data hazards is ensured by the Scheduler, while avoidance of control hazards is ensured by the Reorder Buffer (ROB). Indeed, the Scheduler considers structural hazards before issuing an instruction. The IVM microprocessor has six functional units: Two simple, one complex, one branch and two memory units. Thus, up to six instructions with the above limitations on the type and distribution of instructions can be issued in each clock cycle. Write-After-Read (WAR) and Write-After-Write (WAW) data hazards are taken care of by the Rename module of IVM before the instruction comes to the Scheduler. Read-After-Write (RAW) data hazards, however, may still exist due to dependencies between instruction operands. To deal with such RAW hazards, the Scheduler uses the Scoreboard method [33]. Based on the type of functional unit that will be executing an instruction, the Scoreboard determines the clock cycle in which the destination register for this instruction will be written and available for other instructions to read. Consequently, the Scheduler prevents issuing of instructions that need to use this register prior to the time that it becomes available. Along with each instruction coming to the Scheduler from the Rename module, a unique identification number called ROBid is also provided by the ROB module. This ROBid follows the instruction until it commits and serves as a mechanism for ensuring in-order instruction commitment in the out-of-order execution of the microprocessor and avoiding control hazards.

Fig. 3 shows the high level diagram of the IVM Scheduler module. As shown in this figure, instructions arriving to the Scheduler continue to reside in the buffer even if they have been issued to the Execution unit, until the execution unit confirms that they can start execution. At this time the "valid" and "issued" signals of the related

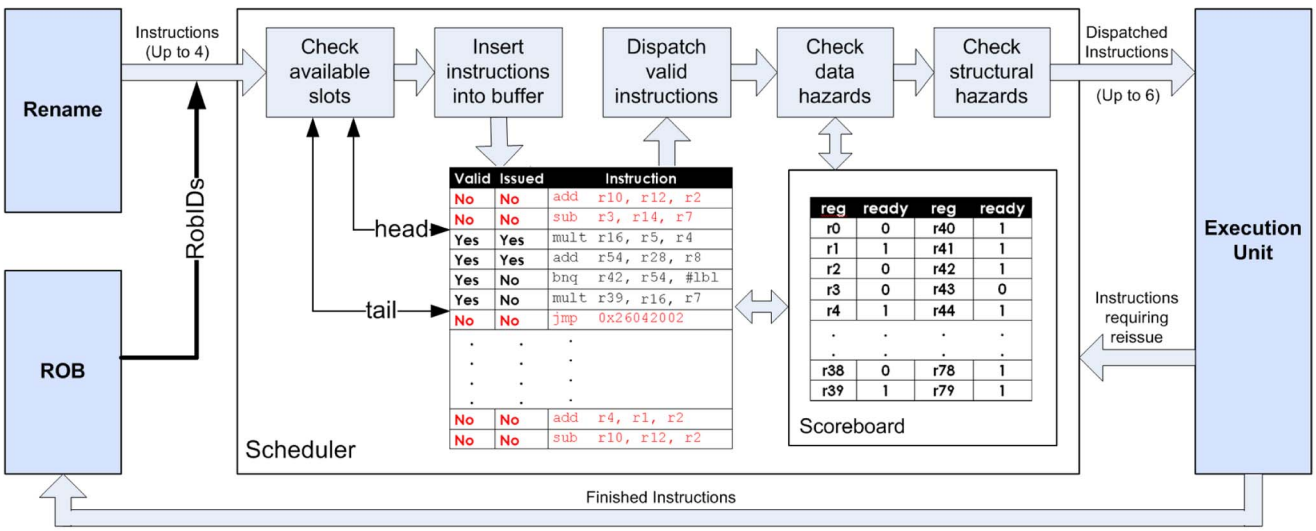

Fig. 3. Block diagram of IVM Scheduler.

TABLE 1
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

instruction in the Scheduler are disabled and the corresponding buffer locations are considered available for use by subsequent instructions.

5 Fault Impact Analysis

The novelty of the CED method proposed herein is that it seeks to leverage information regarding the effect of faults on typical workload. In order to capture this information, we utilize a model of Instruction-Level Errors (ILEs) which was first defined in [29] and which is summarized in Table 1. As shown in this table, ILEs have been divided into five groups reflecting the 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,
3. the functional unit where execution takes place,
4. the starting and finishing time of execution, and
5. the order of commitment.

As part of the simulation infrastructure described in Section 3, we have developed automated software that correlates an injected fault to the ILE that it incurs. We note that the ILE types are not mutually exclusive, i.e., the effect of a fault may be manifested as more than one ILE type. For example, suppose that a fault changes the operation code of an addition instruction to the operation code of a branch instruction; in this case, a Type 1 (Group 1) ILE occurs since an incorrect operation code is used. However, a Type 7 (Group 3) ILE also occurs, since the branch instruction is now executed by a functional unit other than the functional unit used for executing the original addition instruction.

The type of information that we seek in order to support the development of workload-cognizant CED is depicted in Fig. 4. This bar chart summarizes the impact of RT-Level faults in the Scheduler of the IVM microprocessor on the execution of several different SPEC2000 benchmarks, by depicting the distribution of these faults into the various ILE types. The results show a remarkable consistency across the various benchmarks, despite the fact that these benchmarks exercise very different functionality of the microprocessor.

More importantly, the results enable the following observations regarding the occurrence frequency of each ILE type, which drive the development of the CED method described in the next section.

- The majority of faults result in Timing Errors (Group 4) and, in particular, in ILEs of Type 10 (late or no instruction commencement) and Type 11 (longer instruction duration). In other words, ensuring that an instruction is scheduled and executed at the correct time is crucial to the detection of errors.

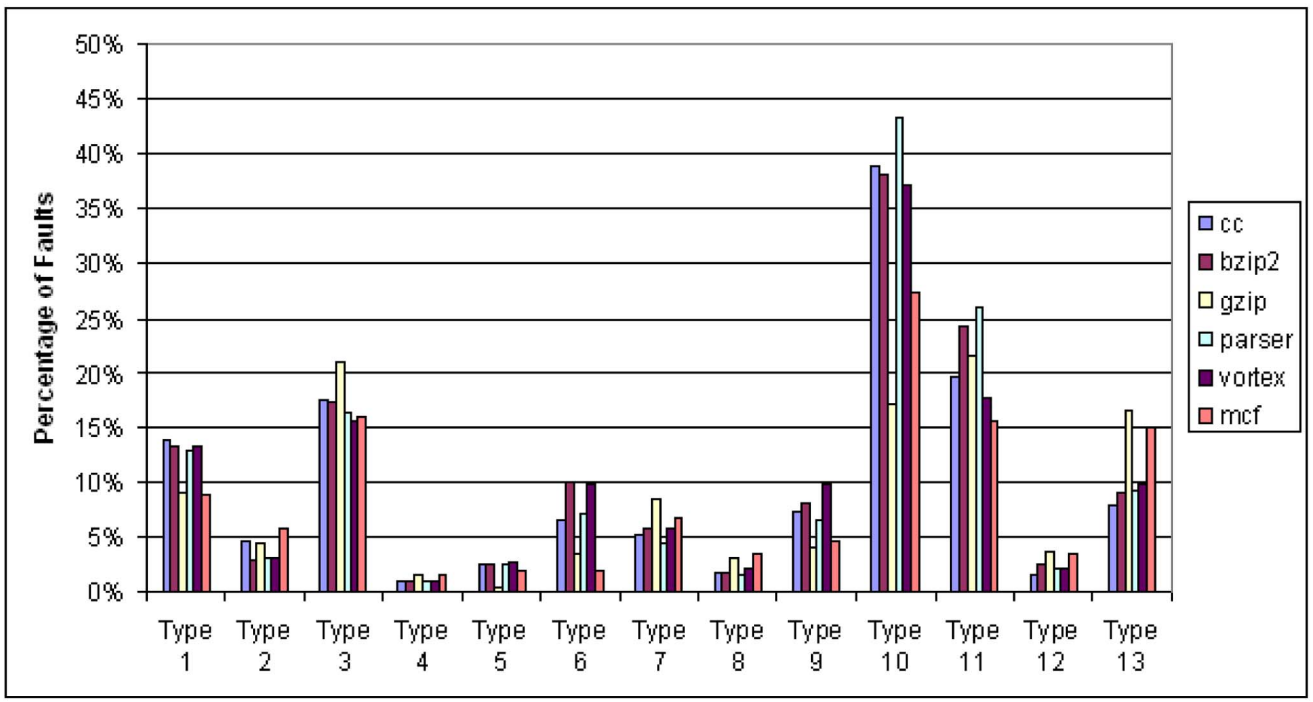

Fig. 4. Distribution of ILE types caused by faults in the Scheduler of IVM.

- A significant number of faults results in a discrepancy in the operands (Group 2) and, in particular, in ILEs of Type 3 (incorrect operand). As a result, ensuring correctness of the utilized resources will contribute a large number of detected errors.
- A tangible number of faults result in a discrepancy in the executed operation (Group 1) and, in particular, in ILEs of Type 1 (incorrect operation code). Hence, ensuring correctness of the executed operations is also important.
- A considerable number of faults result in violation of instruction order (Group 5) and, in particular, in ILEs of Type 13 (commitment order violation). Therefore, ensuring in-order commitment is still a priority but not the most crucial one.
- Only a small number of faults result in discrepancy in the utilized functional unit (Group 3) and, in particular, in ILEs of Type 7 (incorrect functional unit). Thus, ensuring the use of the correct functional unit is less important than the previously mentioned aspects of instruction execution.

6 CED Scheme

Based on the above observations, the CED scheme proposed in this paper seeks to verify the following aspects of instruction execution, which are listed in decreasing order of significance

- Correctness of the time at which an instruction starts executing.
- Correctness of the operands used by the instruction.
- Correctness of the operation code executed by the instruction.
- Correctness of the sequence in which instructions are actually executed, taking into account branches.
- Correctness of the sequence in which instructions arrive to the scheduler.
- Correctness of the functional unit to which an instruction is assigned for execution.

To achieve the above objectives, the proposed CED scheme involves two components. The first component employs additional hardware which predetermines the time at which an instruction should start execution and the functional unit where it should be executed, as explained in Section 6.1. Using this information, the second component employs additional hardware to impose and monitor four invariances that should hold true in each clock cycle i and for each functional unit j,

- **Invariance #1:** Equality between the ROBid of the instruction that is predicted to be executed at time i by functional unit j and the ROBid of the instruction that is actually executed. Monitoring this invariance ensures correct order in the commitment of instructions. Thereby, this invariance aims mainly at detecting ILEs of Group 5 (Order Errors).
- **Invariance #2:** Parity consistency between the operation code of the instruction that is predicted to be executed at time i by functional unit j and the operation code of the instruction that is actually executed. This invariance aims mainly at detecting ILEs of Group 1 (Operation Errors).
- **Invariance #3:** Parity consistency between the operands (up to three) of the instruction that is predicted to be executed at time i by functional unit j and the operands used by the instruction that is actually executed. This invariance aims mainly at detecting ILEs of Group 2 (Operand Errors).
- **Invariance #4:** Parity consistency between the target address of the instruction that is predicted to be executed at time i by functional unit j and the target address of the instruction that is actually executed. Checking this invariance ensures the correctness of execution flow following branch instructions. Thereby, it aims mainly at detecting ILEs of Group 4 (Timing Errors).

We note that no invariance that explicitly checks for ILEs of Group 3 (Execution Errors) is included in our CED scheme. However, such errors are implicitly detected through the prediction of the functional unit where the invariances are checked. Furthermore, as we observed through the fault impact analysis of the previous section, such faults are the least important among the ILE types.

Fig. 5 shows the hardware that needs to be added to the IVM Scheduler in order to support the proposed CED scheme. As we described in Section 4, the Scheduler contains an array where up to 32 instructions coming from the Rename module reside until all structural and data hazards are cleared so that they can be sent for execution. Each entry in this array contains 224 bits comprising the various fields of the instruction, as shown in the figure. Our scheme relies on the construction of a CED Table, which partially replicates this array by keeping only the fields needed for predicting the time and place where an instruction should be executed and supporting the subsequent invariance checking. Specifically, the retained information includes the ROBid, the instruction type, as well as the parity of the operation code, the operands, and the target address if the instruction is a branch. The instruction type, along with information from the existing Scoreboard module and the original instruction array are then used to predict and add to the CED Table the functional unit where an instruction will be dispatched and its execution starting time, expressed as an offset relative to the current clock cycle. All in all, each entry in this Table contains 24 bits.

Given the information in the CED Table, monitoring the four invariances becomes straightforward, as shown in Fig. 6. In each clock cycle, we look at the instruction executed by each functional unit and we extract its ROBid, along with the parity of its operation code, its operands, and its target address. We then compare these fields to the corresponding fields of the instruction that the CED Table predicts as the one that should be executed by this functional unit at this particular time. Any discrepancy signifies an error, in which case the CED output is asserted.

6.1 Predicting Instruction Execution Time and Place

The proposed CED method relies on correctly predicting the time that each instruction will start execution as well as the functional unit to which it will be issued. To achieve this, several parameters need to be considered. First, the availability of resources in the execution unit, which also

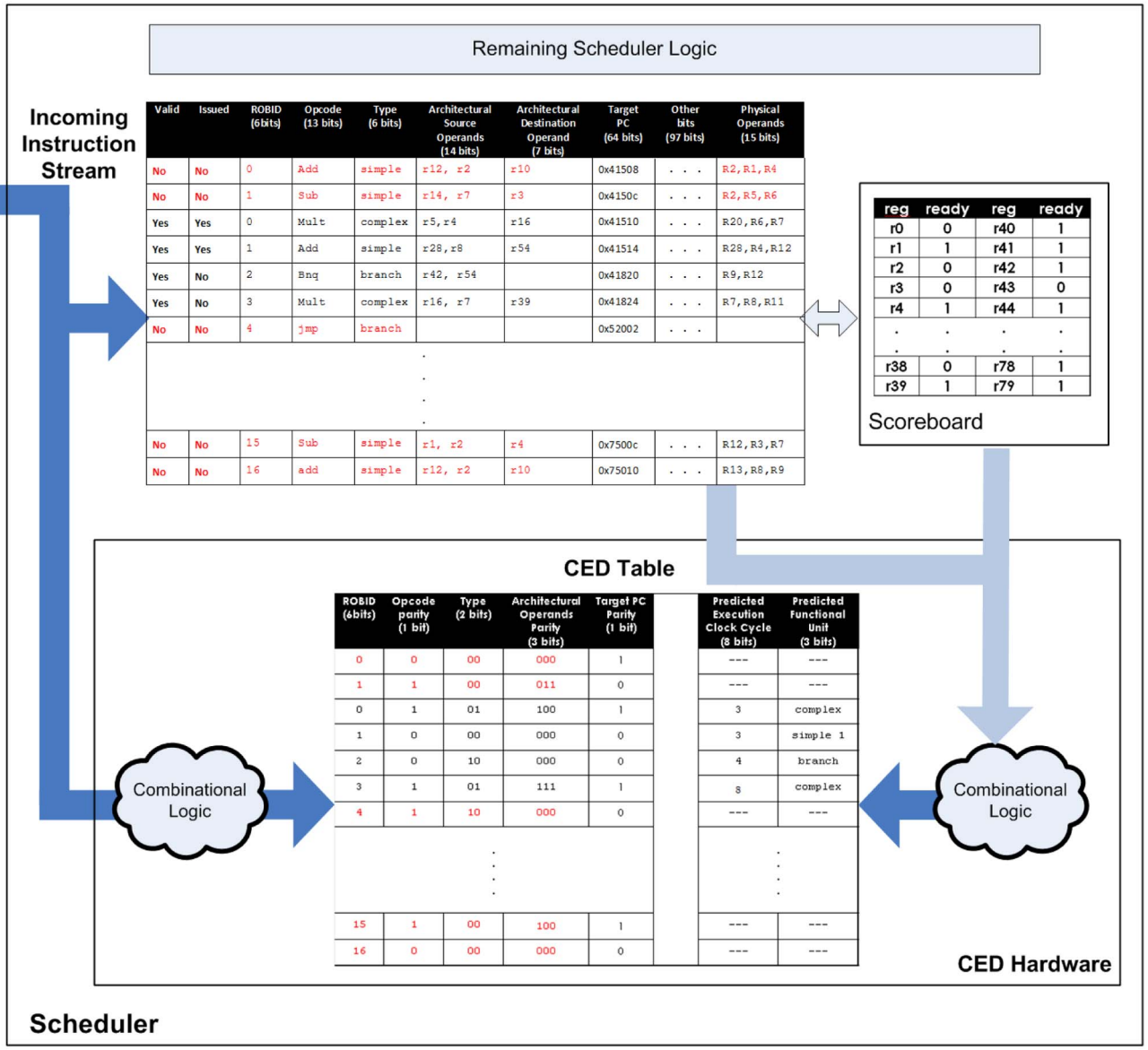

Fig. 5. Hardware additions to the IVM Scheduler to support the proposed CED scheme.

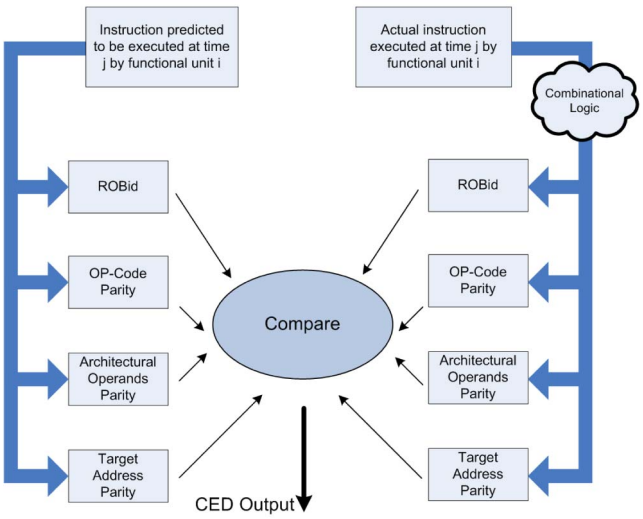

Fig. 6. Invariance checking by proposed CED scheme.

bounds the maximum number and types of instructions that can be issued in each cycle. IVM includes six functional units, namely two simple units, one complex unit, one branch unit, and two memory units. Second, the types and order of arrival of instructions that are waiting to be issued for execution. And, third, the data hazards that an instruction may cause.

Let us consider an instruction that arrives at the Scheduler at clock cycle c. The earliest that this instruction can be issued is at clock cycle $c+1$ and the earliest it can start execution is at clock cycle $c+3$, with the added clock cycles accounting for the pipeline depth between the Scheduler and the Execution Unit. In other words, for the incoming instructions whose issue does not generate any hazard, the execution starting time is determined precisely as soon as the instruction enters the Scheduler. Before an instruction is issued, however, resource availability needs to be considered. Specifically, preceding instructions that are already residing in the Scheduler waiting table are examined

first; if their operands are available and their issuing causes no structural hazards they are given priority over the current instruction, which remains in the Scheduler.

For the instructions which cannot be sent out of the Scheduler in the clock cycle following the clock cycle they enter the Scheduler module, a rough estimate of their execution starting time is made and updated each clock cycle. This estimate depends on the number of instructions of the same type that reside in the Scheduler waiting table, the number of available functional units to execute instructions of this type, as well as the dependencies between the operands of all unissued instructions. In each clock cycle, up to six instructions are issued and the Scheduler waiting table fields are updated. Accordingly, the aforementioned estimates are also updated, until we can determine when all structural and data hazards will be resolved so that we can predict the clock cycle that an instruction will be able to start execution and the functional unit where it will be dispatched. To reduce hardware overhead, the execution starting time of each instruction is expressed as an offset from the current clock cycle.

We also note that although the Scheduler checks the availability of operands before issuing an instruction, a second check takes place in the Execution module. This is required since an operand that was supposed to be available at the required clock cycle (based on the information provided by the Scoreboard), may be not available due to a cache miss. In this case, the Execution unit reports that an instruction needs to be reissued and our method recalculates the new execution starting time.

Example: Consider Fig. 5 which shows a snapshot of the Scheduler waiting table (upper) and the CED table (lower) and let us assume that four instructions arrive at the Scheduler at clock cycle c and are placed in rows 3 through 6 of each table. Let us also assume that no other unissued instructions of these types are waiting in the Scheduler. Based on the instruction types and required operands, our CED scheme determines that the first two instructions (rows 3 and 4) can be issued in the following clock cycle. Taking into account the pipeline depth of three between the Scheduler and the Execution unit of IVM, these two instructions can start execution at clock cycle $c+3$. Accordingly, the offset "3" is placed in the 6th column of rows 3 and 4 in the CED table. However, the branch instruction (row 5) cannot start execution simultaneously with the aforementioned two instructions due to its operand dependency (r54) on the simple instruction (row 4). Hence, our CED scheme consults the Scoreboard regarding the availability time of register r54. Since a simple instruction takes one cycle to execute and the producing instruction starts execution in $c+3$, the Scoreboard responds that r54 will be available in clock cycle $c+4$. Accordingly, clock cycle $c+4$ is predicted as the execution starting time of the branch instruction and the offset "4" is placed in the 6th column of row 5 in the CED table. The next instruction (row 6) is a complex instruction (i.e., multiply) and cannot be immediately issued due to a structural hazard (the first instruction is occupying the complex functional unit). In addition, the first operand of this instruction is r16 which is the destination register of the first instruction (row 3). The multiply unit of the IVM is implemented as a five-stage pipeline, so our CED scheme predicts that the earliest clock cycle at which the instruction in row 6 can start execution is $c+8$. Accordingly, the offset "8" is placed in the 6th column of the row 6 of the CED table.

TABLE 2
Instructions Committed in 2,000 Clock Cycles

Benchmark	cc	bzip2	gzip	mcf	parser	vortex
Instructions	1,058	1,460	1,111	987	2,128	789

7 Results and Analysis

In this section, we provide the details of the simulation setup used for evaluating the effectiveness of the proposed CED scheme, we present the results and we discuss our key observations.

7.1 Simulation Setup

Target Module and Type of Injected Faults: Our CED scheme targets one of the main control modules of the IVM microprocessor, namely the Scheduler, the details of which were presented in Section 4. To assess the efficiency of our CED scheme, we employ the infrastructure described in Section 3 to inject and simulate the effect of single stuck-at faults at the RT-Level description of IVM.[2] There is a total number of 18,822 such faults in the Scheduler of IVM, all of which are used in this study.

Simulation Workload: In order to evaluate the effectiveness of our CED scheme, we use six different SPEC2000 benchmarks as the simulation workload for IVM. Running different benchmarks ensures variability of the instructions executed through the processor and the control logic that they exercise. In each simulation run, the functional simulator is used to execute the first 50,000 clock cycles, thus bypassing the initial system calls and other operations that are not implemented in IVM and reaching a code segment that can be executed on the fault-free RT-Level model of IVM. Then, a fault is injected in the RT-Level model of IVM and each benchmark is executed for 2,000 clock cycles using the RT-Level simulation infrastructure. Table 2 provides the number of instructions committed while running each benchmark for 2,000 clock cycles on the fault-free microprocessor model.

7.2 Experimental Results

We now proceed to present the results of our simulations and to discuss their significance. The results are divided in four sets; the first set provides statistics on fault activation, the second set evaluates the effectiveness of the proposed CED scheme, the third set demonstrates the utility of the impact analysis information based on which the CED scheme was designed, and the fourth set examines various other properties of the CED scheme.

2. In [34], the authors demonstrate a strong correlation between the impact of permanent faults and transient errors, as well as between the impact of Gate-Level and RT-Level faults on instruction execution. Hence, the effectiveness of our CED scheme, as assessed through injection of RT-Level single stuck-at faults, also reflects its effectiveness on Gate-Level faults and transient errors.

TABLE 3
Fault Simulation Outcome After 2,000 Clock Cycles

SPEC Benchmark	Activated Faults	Causing Error	Causing Stall	Causing Exception	Masked Faults
cc	7832 (41.6%)	4180 (22.2%)	2171 (11.5%)	1481 (7.9%)	10990 (58.4%)
bzip2	7997 (42.5%)	4144 (22%)	2217 (11.8%)	1636 (8.7%)	10825 (57.5%)
gzip	5759 (30.6%)	2813 (14.9%)	2269 (12.1%)	677 (3.6%)	13063 (69.4%)
parser	8345 (44.3%)	4591 (24.3%)	2272 (12.1%)	1482 (7.9%)	10477 (55.7%)
vortex	7995 (42.5%)	4483 (23.9%)	2094 (11.1%)	1418 (7.5%)	10827 (57.5%)
mcf	5098 (27.1%)	2844 (15.1%)	1750 (9.3%)	504 (2.7%)	13724 (72.9%)
Average	**7171 (38.1%)**	**3842 (20.4%)**	**2129 (11.3%)**	**1200 (6.4%)**	**11651 (61.9%)**

7.2.1 Fault Activation Profile

Fault simulation outcome: The first set of results, presented in Table 3, summarizes the fault simulation outcome for each benchmark. The second column reports the number of faults that were activated, i.e., they adversely affected the functionality of the microprocessor. On average, 38.1 percent of the faults resulted in a visible discrepancy. In the third through fifth columns, we further split these faults into three categories, based on their impact on instruction execution. As can be observed, an average of 20.4 percent of the activated faults resulted in incorrect architectural state[3] after the executed 2,000 clock cycles, 11.3 percent resulted in a stalling of the pipeline, and 6.4 percent resulted in an exception system call prior to completion of benchmark execution. The final column reports the number of faults that were masked, i.e., they did not affect benchmark execution at all. The high percentage of such errors, averaging at 61.9 percent over the six benchmarks, corroborates our conjecture that application-level error masking plays an important role in modern microprocessors, which motivated the development of the proposed workload-cognizant CED scheme. After all, only a small portion of the functionality of the Scheduler module is exercised by typical workload, hence a large number of faults are either not excited at all or are excited but are logically suppressed.

Fault activation across benchmarks: The second set of results explores the well-known fact that not all faults are equally likely to cause an error (i.e., asymmetric fault activation [8]). Specifically, in Fig. 7 we show the percentage of faults that are activated during the execution of exactly k out of the six benchmarks, $k \in [1, \ldots, 6]$. As may be observed, among the 8,784 faults that are activated in any of the six benchmarks, over 50 percent are activated in all of the six benchmarks and over 85 percent of faults are activated in at least four out of the six benchmarks. The key takeaway point from this observation is that a large number of faults have a high probability of activation independent of the workload that is being executed. Hence, any hardware overhead incurred by a CED scheme that detects these faults is well-justified.

3. The definitions of architectural state and architectural state corruption used herein are borrowed from [27], where it is stated that "In IVM, Microarchitectural state consists of all the SRAM cells, latches, and flip-flops used to implement a processor microarchitecture. Architectural state is a subset of microarchitectural state defined as the state of the machine that is exposed at the instruction set architecture level (e.g., the program counter, register files, and memory state). So Architectural state corruption is any change in PC, register files, and memory state."

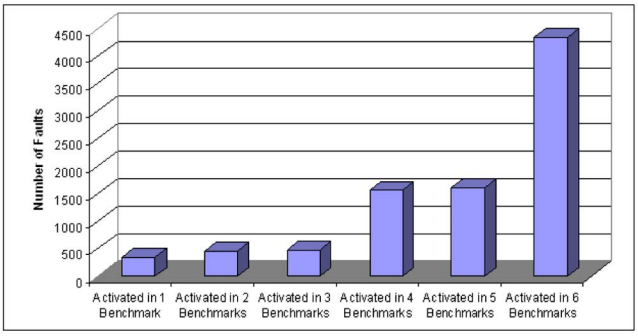

Fig. 7. Faults activated in k benchmarks, $k \in [1, \ldots, 6]$.

7.2.2 CED Effectiveness

CED coverage: The third set of results focuses on the activated faults and reports the effectiveness of the proposed CED scheme in detecting them. Coverage is defined as the percentage of activated faults that are detected by the CED scheme. The attained coverage is shown in Fig. 8. As may be observed, over 85 percent of the faults that affect the architectural state of the processor or prevent it from completing the intended execution of a benchmark due to stalling or exceptions are detected by the proposed CED scheme. Another interesting observation concerns the consistency in the effectiveness of the proposed CED scheme across all six benchmarks, despite the fact that the latter involve execution of different types of instructions and exercise different parts of the functionality of IVM. Indeed, in all cases the achieved fault coverage ranges between 81 percent and 91 percent. Evidently, the CED scheme detects a very large percentage of the activated faults independent of the workload that is being executed.

CED effectiveness across benchmarks: To further demonstrate that the proposed CED is effective across various workloads, in Fig. 9 we break down the number of faults that are activated in k of the six benchmarks, $k \in [1, \ldots, 6]$, based on the number of benchmarks in which they are detected. As may be observed, among the faults

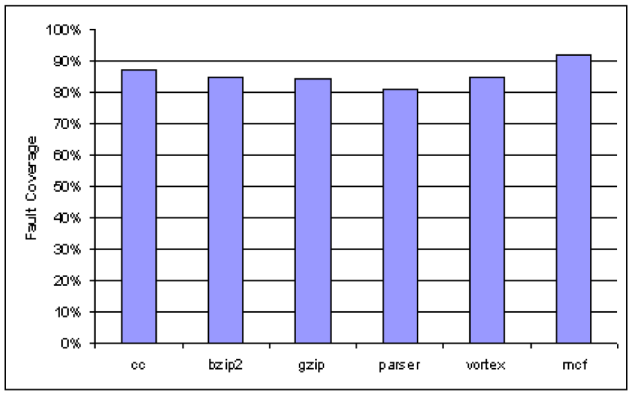

Fig. 8. Coverage of proposed CED scheme.

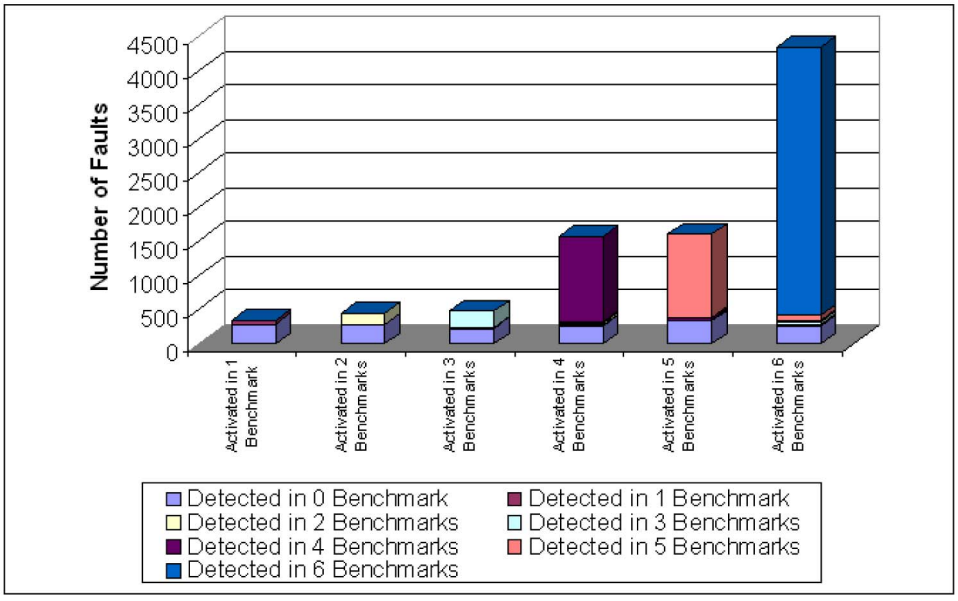

Fig. 9. CED effectiveness in detecting a fault across all benchmarks wherein it is activated.

activated in k benchmarks, the proposed CED scheme indeed detects most of them in all of these k benchmarks. For example, among the faults activated in six benchmarks, over 90 percent are detected in all of these six benchmarks. In other words, the proposed CED scheme detects the targeted faults independent of the workload, thereby benefiting all programs running on the microprocessor and justifying the expended hardware.

Necessity of the four invariances: The next set of results examines whether all four invariances are necessary in the proposed CED scheme. While some overlap between the faults detected by each invariance is expected, this overlap is minimal since each of them mainly targets a different group of ILEs, as discussed in Section 5. To validate this, in Fig. 10 we report the percentage of faults that violate exactly k invariances during the execution of each benchmark, $k \in [1, \ldots, 4]$, averaged over the six benchmarks. As may be observed, 75 percent of the faults violate exactly one invariance and only a small percentage of faults end up simultaneously violating more than one invariances. The key takeaway point of this observation is that the inclusion of each of these invariances in the CED scheme is well-justified, since there is little overlap among the detected fault-sets.

7.2.3 Impact Analysis Utility

Consistency of fault impact across benchmarks: This set of results demonstrates consistency in the instruction-level impact that faults incur, independent of the executed workload. Specifically, Fig. 11 shows the percentage of faults that result in each of the five ILE groups for each of the six benchmarks. As may be observed, the distribution of faults to the five ILE groups is very similar for all six benchmarks, despite the fact that these benchmarks are quite different in the types of instructions they employ and the part of the microprocessor they exercise. This is particularly important because the proposed CED scheme has been developed based on observations regarding the relative importance of the various ILEs caused by low-level faults. Hence, its effectiveness relies on the premise that this distribution is consistent across different workloads.

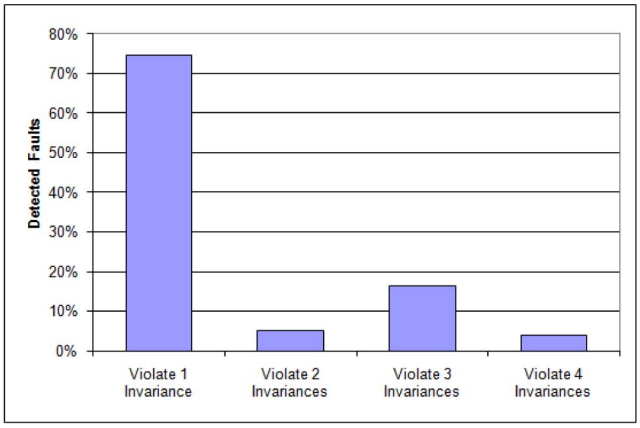

Fig. 10. Faults violating k invariances, $k \in [1, \ldots, 4]$.

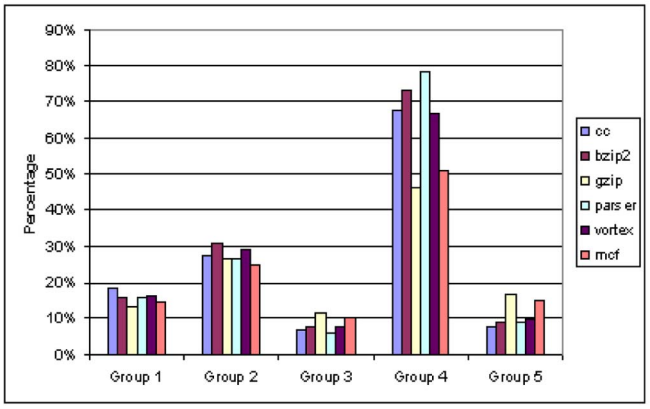

Fig. 11. Percentage of faults causing each ILE group.

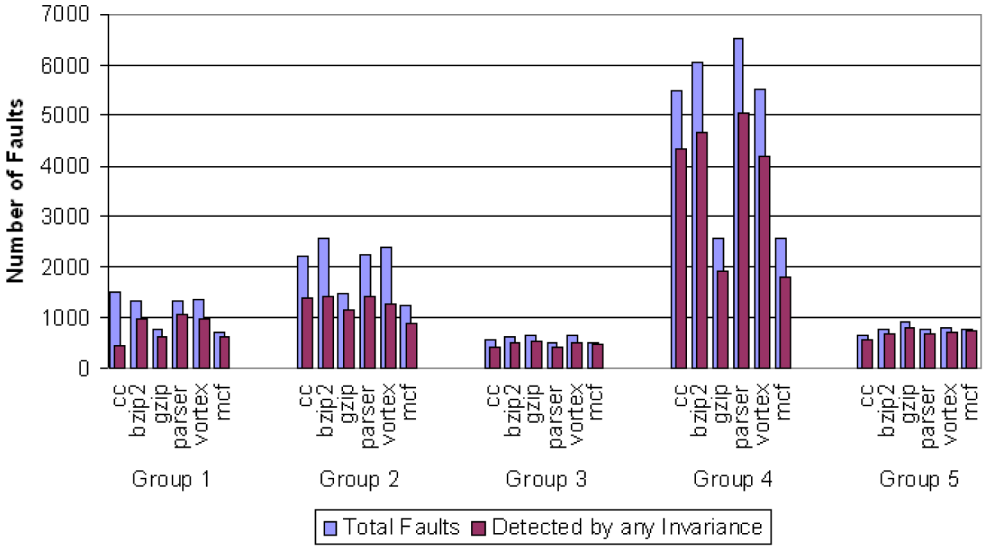

Fig. 12. Total and detected faults causing each ILE group.

CED effectiveness across ILE groups: To further demonstrate the importance of the above consistency, in Fig. 12 we provide the number of faults that result in each ILE group for each benchmark, along with the number of faults detected by the proposed workload-cognizant CED scheme. Evidently, its effectiveness is consistent across the five ILE groups, independent of the workload. Another important observation that can be made using this figure is that most detected faults result in an ILE of group 4. Indeed, as discussed in Section 6, development of the proposed CED scheme was driven by the observation that the dominant ILE types caused by low-level faults are Timing ILEs, which belong to group 4. In other words, the CED scheme was specifically geared towards detecting ILEs of group 4, as validated by the results.

Invariance effectiveness for each ILE group: We now proceed to examine the effectiveness of each of the four invariances included in the proposed CED scheme in detecting the faults that cause each of the five ILE groups. We remind that the four invariances where chosen based on the profile of instruction-level impact caused by low-level faults. Specifically, invariance #1 mainly targets instruction order ILEs (Group 5). Similarly, invariance #2 mainly targets operation ILEs (Group 1), invariance #3 mainly targets operand ILEs (Group 2), and invariance #4 mainly targets timing ILEs (Group 4). Also, execution ILEs (Group 3) are detected as an ancillary benefit of all four invariances. Hence, we expect that each invariance will be the most efficient in detecting the ILEs in the group for which it was designed and may also detect ILEs in other groups but to a lesser extent. Fig. 13 shows the results which validate our expectations. Specifically, based on the results we may observe that each invariance is, indeed, the most effective on the ILE group it was designed for. For example, invariance #4 detects a much higher percentage of faults that result in ILEs of group 4 than the percentage of faults that result in ILEs of the other groups. Furthermore, no other invariance detects more ILEs of a particular group than the invariance that was designed for that ILE group. For example, invariance #4 detects more faults that result in ILEs of group 4 than any of the other invariances. The key takeaway point of these observations is that workload-cognizant analysis of the instruction-level impact caused by low-level faults may effectively guide the selection of appropriate invariances for performing CED.

7.2.4 Other CED Properties

Detection latency: Another question that we seek to address through the next set of results concerns the *detection latency* of the proposed CED scheme. As detection latency, we define the number of clock cycles between the time that an error results in a corruption of the architectural state of the processor and the time that the CED scheme detects the error (i.e., an invariance is violated). The second through fourth columns of Table 4 report the minimum, maximum, and average detection latency of the proposed CED scheme for each benchmark. We note that the detection latency may be negative, implying that the CED scheme alerts to the fact that an error has occurred before this error corrupts the architectural state. The last column of Table 4 reports the percentage of faults that are such *early detections*. Evidently,

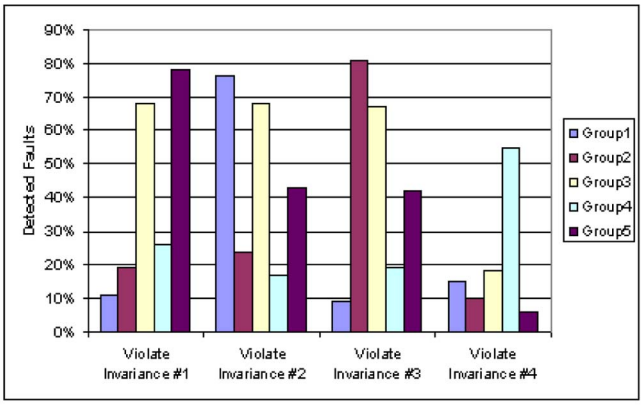

Fig. 13. Effectiveness of invariances on ILE groups.

TABLE 4
Detection Latency of Proposed CED Scheme

SPEC Benchmark	Minimum Latency (clock cycles)	Maximum Latency (clock cycles)	Average Latency (clock cycles)	Early Detections (%)
cc	-1881	1550	4	99.12
bzip2	-1827	795	2	99.52
gzip	-1978	25	1	99.95
parser	-1762	1420	2	99.36
vortex	-1939	1764	2	99.5
mcf	-1786	97	2	99.97
Overall	**-1978**	**1764**	**2**	**99.57**

TABLE 5
Area Overhead Incurred by Proposed CED Scheme

	Number of Cells	Combinational Area (in square microns)	Sequential Area (in square microns)	Total Area (in square microns)
Original Scheduler	92,434	1041434.6	264845.3	1306279.9
Scheduler with Parity-Based CED	134,421	1432132	293431.2	1725563.2
Scheduler with Entire-Field CED	192,493	1903191	394236.4	2297427.4

TABLE 6
Impact on Coverage and Detection Latency Due to the Use of Parity (Masking)

SPEC Benchmark	Activated Faults	Masking in Invariance #2	Masking in Invariance #3	Masking in Invariance #4	Overall Loss of Coverage	Impact on Average Latency (# of cycles)	Impact on Early Detections (%)
cc	7832	2813-2481=332	2946-2678=268	3297-3163=134	6852-6852=0 (0%)	4-4=0	99.12-99.12=0
bzip2	7997	2720-2455=265	2918-2777=141	3227-2843=384	6838-6776=62 (0.77%)	6-2=4	99.52-98.62=0.90
gzip	5759	2151-1991=160	2653-2511=142	1099-1087=12	4870-4870=0 (0%)	1-1=0	99.95-99.95=0
parser	8345	2736-2522=214	2915-2720=195	3027-2761=266	6744-6744=0 (0%)	2-2=0	99.36-99.36=0
vortex	7995	2718-2485=233	2863-2665=198	3097-2889=208	6790-6787=3 (0.03%)	2-2=0	99.54-99.52=0.02
mcf	5098	2119-1874=245	2617-2406=211	1100-1030=70	4673-4673=0 (0%)	2-2=0	99.97-99.97=0

the vast majority of the faults, i.e., over 99.5 percent, are detected early, some of them as early as 1,978 clock cycles before they actually corrupt the architectural state of the processor. Among the rest of the faults, some are not detected until 1,764 clock cycles after they corrupt the architectural state. These, of course, are a few extreme cases. On average, the CED scheme detects faults that are not early detections within a couple of clock cycles. An understanding of the detection latency of the proposed CED scheme is very important since it indicates the window within which the error can be corrected before it affects program execution. While error correction mechanisms are beyond the scope of this paper, one may observe that a simple pipeline flush and restart (which is already supported by IVM) is enough to correct over 99.5 percent of the faults (i.e., all the early detections). For the rest of the faults, which are detected after the architectural state is corrupted, the average detection latency of two clock cycles implies that simple checkpoint-and-restore operations covering a small window of a few clock cycles would be capable of correcting them.

Incurred area overhead: The next set of results focuses on the cost-effectiveness of the proposed method. To this end, we converted the Scheduler module of IVM to synthesizable Verilog and we synthesized the Scheduler module before and after applying our CED scheme using Synopsys Design Compiler and targeting a TSMC .13 micron library. The first two rows of Table 5 show the area of the combinational and sequential parts of the Scheduler before and after inclusion of our parity-based CED scheme. As shown in this table, the incurred area overhead of the CED scheme is 32 percent of the area of the Scheduler, making it an attractive proposition considering that it covers over 85 percent of the activated faults and with a detection latency of only a few cycles. While this overhead figure does not include any additional area needed for signal routing, we point out that the corresponding effectiveness also does not include faults in other modules that are detected as an ancillary benefit of the proposed CED scheme.

Masking due to parity: The final set of results assess the limitations of using parity for invariances #2, #3, and #4, as opposed to comparison of the entire fields (we remind that invariance 1 is based on the entire ROBid field). Specifically, Table 6 reports the number of faults that would be detected by comparing the entire fields but are not detected by comparing the parity in each of these three invariances, as well as the overall coverage loss due to masking. As may be

observed, while the coverage of each individual invariance takes a light hit, the fact that faults may be detected by more than one invariance compensates for this effect, resulting in only a small 0.77 percent loss in `bzip2`, a negligible 0.03 percent loss in `vortex` and no coverage loss in the remaining four benchmarks. Moreover, as may be observed in the last two columns of Table 6, the impact on average detection latency and early detection is also negligible. Also, as indicated in the third row of Table 5, the area overhead would 2.37 times higher if the CED scheme compares the entire fields instead of the parity (75.8 percent as opposed to 32 percent). These observations reveal that parity is a cost-effective option.

8 Conclusion

Modern microprocessors exhibit a high degree of application-level error masking, which prevents a significant percentage of faults occurring in the field of operation from corrupting their architectural state. The conjecture supported by the work presented herein is that one may leverage this asymmetric criticality of faults in order to develop cost-effective CED methods. Indeed, a careful analysis of the impact of such faults on typical microprocessor workload provides the basis for identifying the most vulnerable functionality of the microprocessor and pinpoints the areas where CED resources should be expended. To demonstrate this principle, we investigated the prevalent instruction-level errors that are caused by faults in the Scheduler of the IVM microprocessor and we developed a number of invariances which monitor the most important aspects of instruction execution. Thereby, we constructed a CED scheme which only takes up 32 percent of the Scheduler area and which is capable of detecting over 85 percent of the Scheduler faults that affect the architectural state of the processor while the latter is executing SPEC2000 benchmarks. Combined with the observation that most faults are detected prior to corrupting the architectural state, and that the few that do corrupt the architectural state are detected within a few clock cycles, the above result makes this workload-cognizant CED method a competitive proposition.

Acknowledgments

This work was supported by a generous gift from the Intel Corp. The first two authors contributed equally to this work. The first author performed this research while being a visiting student at Yale University. A preliminary version of part of the results reported herein was presented at the 2008 International Symposium on Defect and Fault Tolerance in VLSI Systems [35]. 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] S. Matakias, Y. Tsiatouhas, A. Arapoyanni, and T. Haniotakis, "A Circuit for Concurrent Detection of Soft and Timing Errors in Digital CMOS ICs," *J. Electronic Testing: Theory and Applications,* vol. 20, no. 5, pp. 523-531, 2004.

[2] P. Hazucha, C. Svensson, and S.A. Wender, "Cosmic-Ray Soft Error Characterization of a Standard 0.6μm CMOS Process," *IEEE J. Solid-State Circuits,* vol. 35, no. 10, pp. 1422-1429, Oct. 2000.

[3] E. Normand, "Single Event Upset at Ground Level," *IEEE Trans. Nuclear Science,* vol. 43, no. 6, pp. 2742-2750, Dec. 1996.

[4] Y. Tosaka, S. Satoh, T. Itakura, H. Ehara, T. Ueda, G.A. Woffinden, and S.A. Wender, "Measurement and Analysis of Neutron-Induced Soft Errors in Sub-Half-Micron CMOS Circuits," *IEEE Trans. Electron Devices,* vol. 45, no. 7, pp. 1453-1458, July 1998.

[5] C. Metra, M. Favalli, and B. Ricco, "On-Line Detection of Logic Errors Due to Crosstalk, Delay, and Transient Faults," *Proc. Int'l Test Conf.,* pp. 524-533, 1998.

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

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

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

[9] S. Mitra and E.J. McCluskey, "Design Diversity for Concurrent Error Detection in Sequential Logic Circuits," *Proc. Very Large Scale Integration (VLSI) Test Symp.,* pp. 178-183, 2001.

[10] A. Avizienis and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer,* vol. 17, no. 8, pp. 67-80, 1984.

[11] G. Aksenova and E. Sogomonyan, "Design of Self-Checking Built-in Check Circuits for Automata with Memory," *Automation and Remote Control,* vol. 36, no. 7, pp. 1169-1177, 1975.

[12] S. Dhawan and R.C. De Vries, "Design of Self-Checking Sequential Machines," *IEEE Trans. Computers,* vol. 37, no. 10, pp. 1280-1284, Oct. 1988.

[13] C. Zeng, N. Saxena, and E.J. McCluskey, "Finite State Machine Synthesis with Concurrent Error Detection," *Proc. Int'l Test Conf.,* pp. 672-679, 1999.

[14] M. Pflanz, K. Walther, C. Galke, and H.T. Vierhaus, "On-Line Error Detection and Correction in Storage Elements with Cross-Parity Check," *Proc. Int'l On-Line Test Workshop,* pp. 69-73, 2002.

[15] D. Das and N.A. Touba, "Synthesis of Circuits with Low-Cost Concurrent Error Detection Based on Bose-Lin Codes," *Proc. Very Large Scale Integration (VLSI) Test Symp.,* pp. 309-315, 1998.

[16] N.K. Jha and S.-J Wang, "Design and Synthesis of Self-Checking VLSI Circuits," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 12, no. 6, pp. 878-887, June 1993.

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

[18] J.H. Patel and L.Y. Fung, "Concurrent Error Detection in ALUs by Recomputing with Shifted Operands," *IEEE Trans. Computers,* vol. 31, no. 7, pp. 589-595, July 1982.

[19] S. Almukhaizim, P. Drineas, and Y. Makris, "On Concurrent Error Detection with Bounded Latency in FSMs," *Proc. Conf. Design, Automation and Test,* vol. 1, pp. 596-601, 2004.

[20] R. Vemu, A. Jas, J.A. Abraham, S. Patil, and R. Galivanche, "A Low-Cost Concurrent Error Detection Technique for Processor Control Logic," *Proc. Conf. Design, Automation and Test in Europe,* pp. 897-902, 2008.

[21] Y. Makris, I. Bayraktaroglu, and A. Orailoglu, "Enhancing Reliability of RTL Controller-Datapath Circuits via Invariant-Based Concurrent Test," *IEEE Trans. Reliability,* vol. 53, no. 2, pp. 269-278, June 2004.

[22] 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.

[23] A. Mahmood and E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors—A Survey," *IEEE Trans. Computers,* vol. 37, no. 2, pp. 160-174, Feb. 1988.

[24] M. Jafari-Nodoushan, S. Ghassem-Meremadi, and A. Ejlali, "Control Flow Checking Using Branch Instructions," *Proc. Int'l Conf. Embedded and Ubiquitous Computing,* pp. 66-72, 2008.

[25] A. Mendelson and N. Suri, "Designing High-Performance and Reliable Superscalar Architectures—The Out of Order Reliable Superscalar (O3RS) Approach," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 5-28, 2000.

[26] J.B. Nickel and A.K. Somani, "REESE: A Method of Soft Error Detection in Microprocessors," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 401-410, 2001.
[27] 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.
[28] 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," *Proc. Int'l Symp. Microarchitecture,* pp. 29-40, 2003.
[29] N. Karimi, M. Maniatakos, Y. Makris, and A. Jas, "On the Correlation between Controller Faults and Instruction-Level Errors in Modern Microprocessors," *Proc. Int'l Test Conf.,* pp. 24.1.1-24.1.10, 2008.
[30] 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.
[31] D. Burger, T.M. Austin, and S. Bennett, "Evaluating Future Microprocessors: The Simplescalar Tool Set," Technical Report CS-TR-1996-1308, Intel Corporation, 1996.
[32] J.C. Baraza, J. Garcia, 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),* vol. 16, no. 6, pp. 693-706, June 2008.
[33] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* third ed. Morgan Kaufmann, 2003.
[34] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and Y. Makris, "Instruction-Level Impact Analysis of Low-Level Faults in a Modern Microprocessor Controller," *IEEE Trans. Computers,* vol. 60, no. 9, pp. 1260-1273, 2011.
[35] 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. Int'l Symp. Defect and Fault Tolerance in VLSI Systems,* pp. 454-462, 2008.

Naghmeh Karimi received the BS, MS, and PhD degrees in computer engineering from the University of Tehran, Iran, in 1997, 2002, and 2010, respectively. Her masters thesis was on testability enhancement at the Register Transfer Level and her 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.

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, Connecticut, in 2008, where he is currently working toward his PhD degree. His current research interests include test and reliability of modern microprocessors and computer architecture. He is a student member of the IEEE.

Abhijit Jas received the BE degree in computer science and engineering from Jadavpur University, Kolkata, India, in 1996, and the MS and PhD degrees in electrical and computer engineering from the University of Texas at Austin in 1999 and 2001, respectively. He secured the first rank among all graduating students from the college of engineering. He is currently working as a component design engineer with the Design and Technology 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.

Chandrasekharan (Chandra) Tirumurti is a research scientist with the Design and Technology Solutions group at Intel Corporation based in Santa Clara, California. His current focus is on strategic manufacturing test initiatives for mainstream CPUs. As 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 the Intel and is an avid cricketer. He is a member of the IEEE.

Yiorgos Makris received the diploma degree in 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.