J Electron Test (2013) 29:351–366
DOI 10.1007/s10836-013-5360-5

On the Impact of Performance Faults in Modern Microprocessors

Naghmeh Karimi · Michail Maniatakos · Chandrasekharan (Chandra) Tirumurti · Yiorgos Makris

Received: 12 March 2012 / Accepted: 7 February 2013 / Published online: 21 March 2013

Abstract Modern microprocessors incorporate a variety of architectural features, such as branch prediction and speculative execution, which are not critical to the correctness of their operation yet are essential towards improving performance. Accordingly, while faults in the corresponding hardware may not necessarily affect functional correctness, they may, nevertheless, adversely impact performance. In this paper, we investigate quantitatively the performance impact of such faults using a superscalar, dynamically-scheduled, out-of-order, Alpha-like microprocessor, on which we execute SPEC2000 integer benchmarks. We provide extensive fault simulation-based experimental results that elucidate the various aspects of performance faults and we discuss how this information may guide the inclusion of additional hardware for performance loss recovery and yield enhancement.

Keywords Modern microprocessors · Performance faults · Architectural features · Performance loss recovery · Yield enhancement

1 Introduction

In their quest towards maximizing instruction level parallelism (ILP) and, thereby, improving performance, computer architects have equipped modern microprocessors with an impressive arsenal of advanced features. Superscalar machines, advanced cache management strategies, data pre-fetching, data value and branch prediction are only a few examples of such techniques [1–6]. Out-of-order instruction execution capabilities, in particular, combined with advanced multi-level branch prediction schemes, play a crucial role in today's state-of-the-art, deeply-pipelined, speculative processors [7–9]. Interestingly, many of these architectural features are geared solely towards performance improvement and their presence is not critical to the correctness of execution. As a result, potential malfunctions in the corresponding hardware may not jeopardize the outcome of the workload executed by a microprocessor in any way other than simply delaying it [10–12]. Hence, in this work, we will refer to faults resulting in such benign malfunctions as *performance faults*.

The research reported herein aims to investigate the impact that such performance faults may have on the execution of typical microprocessor workload. To this end, we employ the Register Transfer (RT-) Level model of an Alpha-like microprocessor exhibiting most of the aforementioned advanced architectural features, on which we simulate execution of SPEC2000 benchmark programs. Specifically, we seek to quantify the number of faults that cause no functional discrepancy but only reduce

Responsible Editor: M. Violante

N. Karimi (✉)
ECE Department, Polytechnic Institute of New York University,
Brooklyn, NY 11201, USA
e-mail: nkarimi@poly.edu

M. Maniatakos
ECE Department, New York University, Abu Dhabi,
PO Box 129188, Abu Dhabi, UAE
e-mail: michail.maniatakos@nyu.edu

C. Tirumurti
SoC Enabling Group, Intel Corporation,
Santa Clara, CA 95050, USA
e-mail: chandra.tirumurti@intel.com

Y. Makris
EE Department, The University of Texas at Dallas,
Richardson, TX 75080, USA
e-mail: yiorgos.makris@utdallas.edu

performance, as well as the level of the incurred performance degradation. Furthermore, we are interested in assessing the relative importance of performance faults across various workloads. Such information can potentially guide the addition of hardware or the development of software-based self-testing (SBST) [13] schemes to alleviate the most crucial performance faults and recover the lost performance, or even to enhance yield by adding hardware that converts actual functionality faults into performance faults. In fact, SBST schemes have been previously proposed [14–16] to detect performance faults in the branch prediction unit of processors. The work described herein, however, extends the focus of studying the impact of performance faults beyond the branch predictor, covering three key modules of a modern microprocessor.

The rest of this paper is organized as follows. In Section 2, we take a closer look at the concept of performance faults and the underlying architectural features that facilitate their existence. Then, in Section 3, we describe the microprocessor model which serves as a vehicle for this study and we provide a detailed description of the target modules. In Section 4, we discuss the capabilities of the simulation infrastructure which is used in this study. The employed performance impact analysis method is detailed in Section 5 and extensive results quantifying the impact of performance faults are presented in Section 6. Detailed examples of performance faults are given in Section 7. In Section 8, we discuss the potential utility of this study in guiding hardware addition for performance loss recovery and yield enhancement. Conclusions are drawn in Section 9.

2 Performance Faults

Among the various architectural concepts that bring about the class of performance faults, we pinpoint three prominent ones: pipelining, superscalar design, and speculative execution.

Instead of waiting for all necessary resources to become available prior to execution of an instruction, pipelining allows some of the involved tasks to be completed early. Thus, resources that would otherwise be idle are utilized and, subsequently, freed for use by other instructions, increasing the overall performance. Faults which prevent this early utilization of resources may not cause incorrect results but will reduce the throughput, hence incurring performance degradation.

Superscalar processors increase performance by employing multiple functional units, often even of the same type, in order to execute many instructions in parallel. Intricate hardware-implemented algorithms are, consequently, employed to optimally schedule execution of instructions by these functional units. Hence, faults interfering with this process may result in a suboptimal scheduling of instructions that yields a correct, yet performance-impacted execution.

Speculative execution aims to maximize performance by leveraging resources that would otherwise be idle and allowing instructions to proceed with execution even though validity of the corresponding resources, or even the instructions themselves, is yet to be determined. Along with it comes an inherent mechanism for discarding the speculatively executed instructions in case the speculation proves to be incorrect. Speculation happens in various aspects of modern microprocessors, involving control, data, or both [17, 18]. The most common forms of speculation are those predicting the direction of program control, particularly involving prediction of the direction of branch instructions. A number of data speculation mechanisms, such as value prediction (e.g. index counter variables), address prediction (e.g. addresses of array elements) and memory system optimism (e.g. returning a value from a cache before checking its validity) are also frequently employed Given the speculative nature of these architectural features and the inherent recovery mechanism, it is evident that faults interfering with this process may not affect execution correctness but will impact performance, sometimes even positively!

3 Study Framework

Our investigation on the impact of performance faults builds upon a previously developed fault simulation infrastructure, which is presented in detail in [19]. The employed model is the Verilog implementation of an Alpha-like microprocessor, called IVM (Illinois Verilog Model) [20, 21]. IVM implements a subset of the instruction set of the Alpha 21264 microprocessor. Consisting of approximately 40,000 state elements, the IVM is rich in architectural features including: superscalar, out-of-order execution, dynamically scheduled pipeline, hybrid branch prediction and speculative instruction execution. IVM can have up to 132 instructions in-flight through its 12-stage pipeline, supported by a dynamic scheduler of 32 entries and 6 functional units. The complexity of IVM reflects most of the features of modern, high-performance microprocessors. Furthermore, it allows simulation of the execution of actual workload, such as the SPEC2000 benchmarks. Thus, it enables a realistic investigation of the impact of performance faults in 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 [22]. 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. Figure 1 shows the block diagram of IVM, as presented in [20].

In this research, we focus on three key modules of the IVM microprocessor, namely the Scheduler, the ReOrder Buffer (ROB), and the Fetch Unit. The following subsections provide details on the functionality of these modules, along with a discussion on how this functionality lends itself to the existence of performance faults.

3.1 Scheduler

Functionality The Scheduler is a dynamic module which contains an array of up to 32 instructions waiting to be issued. Each instruction coming to the Scheduler resides in this buffer until an acknowledgement 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. Instructions are issued out of order depending on the following factors:

- Availability of instructions in the Scheduler
- Avoidance of structural hazards
- Avoidance of data hazards

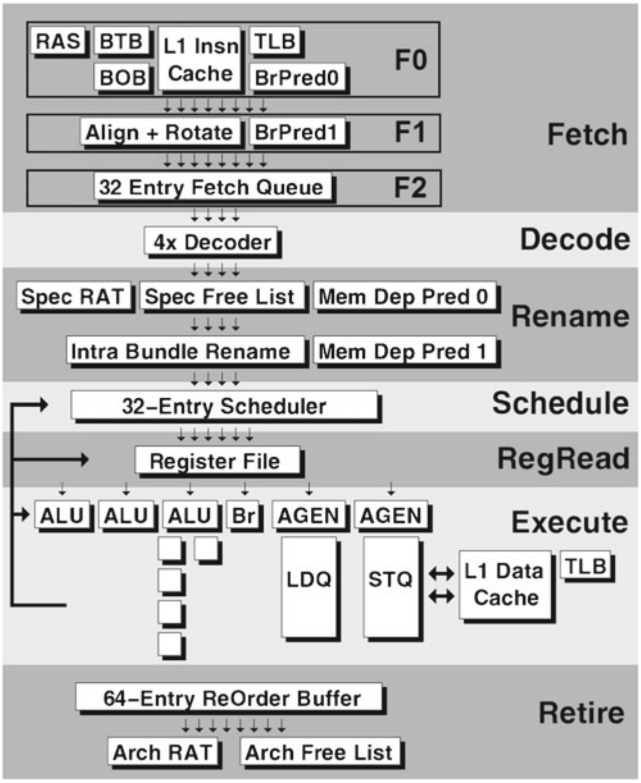


Fig. 1 IVM block diagram [20]

Structural hazards are considered by the Scheduler before issuing an instruction. The IVM microprocessor has 6 functional units: 2 simple, 1 complex, 1 branch and 2 memory units. Thus, up to 6 instructions with the above limitations on the type 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. 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 [23]. Based on the type of functional unit that will be executing an instruction, the Scoreboard determines when 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.

Possible Performance Faults Any fault that makes a register appear to be ready for reading prior to its time of availability will make the processor issue the dependent instructions before their operands are actually ready. Nevertheless, readiness of registers is checked again in the IVM Execution unit before instruction execution commences; thus, if such a fault occurs, the execution unit makes the Scheduler reissue the instruction until the operands are really ready. As a result, such faults only affect the performance of the processor but not the correctness of its functionality. Similarly, if a fault in the scoreboard prolongs the unavailable window for reading a register, dependent instructions are issued with a delay which affects performance but not correctness.

In addition, as shown in Fig. 2, in order to execute an instruction the Scheduler sends the instruction, along with an ‘issue’ and a ‘valid’ signal, to the corresponding functional unit. The former indicates whether the instruction has been issued to a functional unit, while the latter indicates validity of the instruction. Both signals are set by the Scheduler. When an instruction is issued, the readiness of the corresponding operands is checked and two acknowledge signals are sent back to the Scheduler for each issued instruction. In fact, if for any reason the functional unit can not start execution, the scheduler re-issues the instruction. Hence, many faults disrupting this handshaking protocol will only result in performance degradation but not in erroneous functionality.

3.2 ReOrder Buffer

Functionality In a processor with out-of-order execution capabilities, a ReOrder Buffer (ROB) module serves as the mechanism to ensure in-order instruction commitment. This

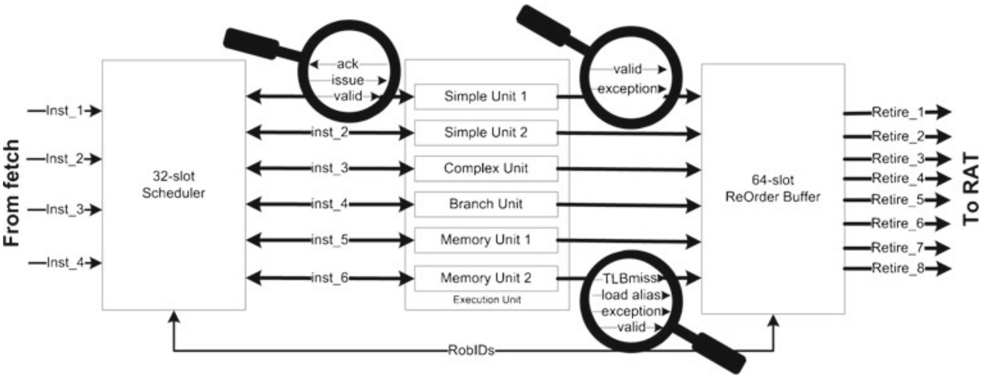


Fig. 2 Instruction execution flow in IVM

is achieved by means of an identification number, called ROBid, which the ROB assigns to each instruction that comes to the Scheduler and which follows the instruction until it commits. In IVM, the ROB has a queue of 64 locations, each of which corresponds to a dispatched instruction that has yet to be committed. Instructions proceed from IVM's execution unit to the ROB, where up to 8 instructions can be retired in each clock cycle, after a number of factors are considered.

Specifically, upon completion of execution of an instruction, a number of signals are sent to the ROB from the corresponding functional unit, as shown in Fig. 2. These signals include the ROBid of the instruction, a 'Valid' signal signifying execution completion, an 'Exception Reporting' signal which indicates whether any exception was raised during execution of a non-branch instruction, as well as a 'Translation Lookaside Buffer (TLB) Miss' and a 'Load Aliasing' signal for Store and Load instructions. Based on this information, the ROB ensures correct commitment order by retiring Instruction J ($inst_J$) in clock cycle C if the following conditions hold:

- No exception was raised during execution of $inst_J$
- If $inst_J$ is either a Store or a Load instruction, no TLB miss occurred
- If $inst_J$ is a Load instruction, no Load Aliasing occurred
- Potential flush due to a mispredicted branch cannot remove $inst_J$ from the ROB prior to its commitment
- Preceding instructions (i.e. dispatched with earlier ROBid than $inst_J$) have either been retired in previous clock cycles or retire in clock cycle C
- Only one branch instruction can be retired in a clock cycle. If there are more branch instructions among those that are ready to retire in clock cycle C, only the instructions up to the second branch are retired.

Possible Performance Faults The ROB compares the target Program Counter (PC) of each branch instruction to the PC of its next instruction. In case of a discrepancy, the pipeline is flushed after retiring the branch instruction and the target PC is sent to the Fetch Unit to show the address from which the instructions should be fetched. Therefore, any fault in the ROB that results in an unintended branch flush, will result in fetching and executing the flushed instructions again, thus decreasing the performance and throughput of the pipeline.

In addition, the ROB sends a 'Direction' signal to the Fetch Unit, indicating the outcome of the last retired branch instruction (Taken or Not Taken). This signal is used by the Fetch Unit, along with other parameters, to predict the direction of the following branch instructions. Any fault that changes this signal may affect the stream of instructions that follow the next branch instruction and may result in unintended pipeline flushes which, in turn, will affect the performance of the processor.

3.3 Fetch Unit

Functionality Speculative processors fetch instructions into the pipeline before it is known whether they should be executed or not. When a branch instruction is fetched, its direction (Taken or Not Taken) is predicted immediately so that more instructions can be fetched, but the actual direction status is only determined after it is executed. At that point, if the prediction was incorrect, the pipeline is flushed and the Fetch Unit resumes fetching instructions from the correct address [24]. The IVM Fetch Unit fetches 4 adjacent instructions by reading two lines from the instruction cache,[1] and delivers them to the Decoder. Fetch has 3 phases, F0, F1, and F2, shown in Fig. 1.

F0 looks up the Branch Target Buffer (BTB) and Return Address Stack, and makes a simple branch prediction which feeds F1. In addition, F0 starts the first stage of the meta

[1] The instruction cache stores four 32-bit instructions per line, aligned to 16 bytes. However, the first instruction to be fetched in each cycle may not be aligned on a 16-byte boundary. Thus, to guarantee delivery of 4 instructions, two cache lines need to be accessed.

branch prediction. The BTB is structured as a multi-way associative cache and is looked-up via a portion of the PC. Note that all BTB ways are accessed simultaneously and their tags are compared to the PC to determine whether the data in any of the ways is a match. The predictor in the BTB is per BTB entry (i.e. per branch). In IVM, each BTB has 256 two-bit locations.

F1, which has access to the instruction cache, completes the meta branch prediction and inserts the instructions fetched from the instruction cache into the fetch queue. If the meta branch prediction does not match the simple branch prediction from the BTB, the Fetch Unit performs a mini pipeline flush, redirecting F0 to the address predicted by the meta branch predictor and invalidating the instructions that were already fetched after the branch.

F2, equipped with a 32 location queue, receives up to 8 instructions in each clock cycle and sends up to 4 instructions to the decoder. In essence, F2, decouples the Fetch Unit from the rest of the processor.

Possible Performance Faults Any malfunction of the simple branch predictor in F0 or the meta branch predictor in F1 may cause an unexpected pipeline flush, either in phase F1 or later in the ROB after the branch instruction is executed, causing performance degradation. Of course, the performance lost by an unnecessary pipeline flush in the ROB is more significant, as compared to the pipeline flush in the F1 phase.

In addition, since the BTB contents are used to predict the branch target address when a branch is predicted as Taken, faults in the BTB may result in considerable performance loss. Even if the branch is, indeed, correctly predicted as Taken, a fault in the BTB contents will lead the processor to execute an incorrect stream of instructions and will finally result in a pipeline flush after the related branch instruction is executed and its correct target address is determined. Since this pipeline flush will not happen until the branch instruction is retired in the ROB, and considering the out-of-order instruction execution capability of IVM, flushing the pipeline may result in dumping many executed instructions from the pipeline and consuming many clock cycles to execute the correct instructions.

4 Simulation Capabilities

One of the most valuable capabilities of IVM and its complementary functional simulator is their ability to execute SPEC2000 benchmarks, thus allowing simulation of real workload. However, the IVM version that has this capability is, unfortunately, not synthesizable. Thus, we cannot make use of gate-level fault simulation tools for the purpose of this study. Instead, we employ an RT-Level fault simulator[2] which was developed and presented in detail in [19], wherein fault injection is performed using a method similar to the parallel saboteurs technique described in [26]. 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.

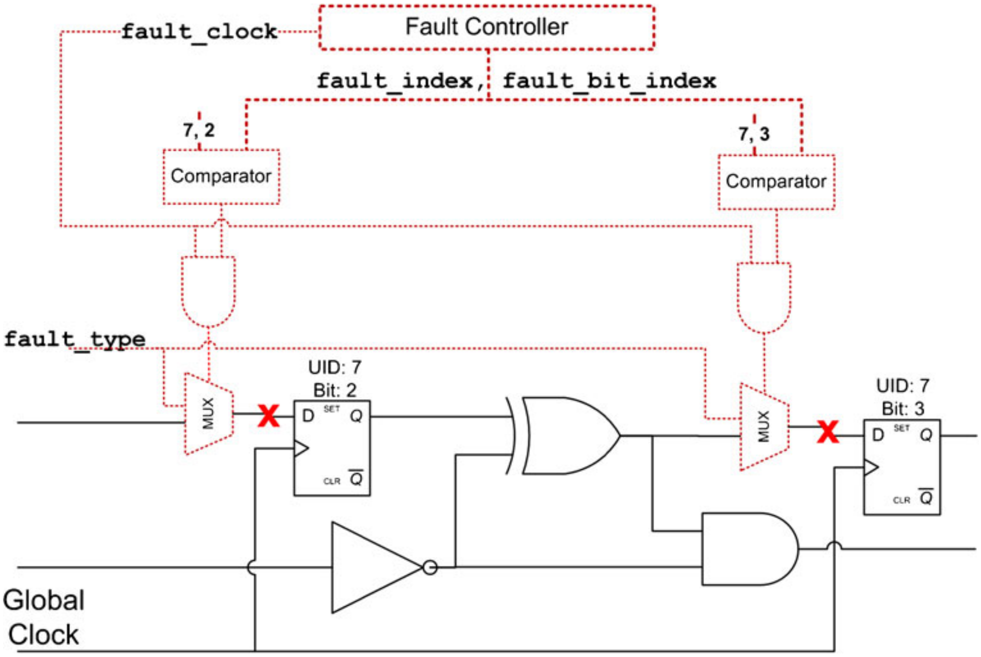
Figure 3 shows a high-level diagram of this method, which allows injection of either stuck-at or transient faults with user-defined activation times to any entity defined in the RT-Level Verilog model (dotted lines indicate hardware added for fault simulation). As shown in this figure, each storage element 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.

5 Performance Impact Analysis Method

The aforementioned simulation capabilities provide us with the necessary infrastructure to evaluate the impact of faults in key speculative execution modules of a modern microprocessor on its performance. Our main objective is to gain insight regarding the extent of the problem, including both the number of performance faults and the level of performance degradation that they incur. To this end, we employ the fault simulation-based performance evaluation approach depicted in Fig. 4. Specifically, we start by simulating the execution of k instructions of a typical workload and recording the corresponding number of clock cycles, $CC_g(k)$,

[2] While our performance impact analysis is performed at the RT-Level, we note that a recent study reveals a very strong correlation between the impact of RT-Level and Gate-Level faults on the execution of workload in the IVM processor [25]; hence, we expect that the obtained results are representative of what would be obtained through gate-level fault simulation.

Fig. 3 RT-level fault injection method



along with the resulting architectural state,[3] $AS_g(k)$, and the machine state, $MS_g(k)$, of the golden (fault-free) microprocessor model. We, then, repeat the simulation injecting one fault at a time and recording the corresponding number of clock cycles, $CC_f(k)$, along with the resulting architectural state, $AS_f(k)$, and the machine state, $MS_f(k)$, of the faulty microprocessor model. A comparison between the simulation outcomes classifies each fault to one of the following types:

- **Functionality Fault:** A difference between the architectural states $AS_g(k)$ and $AS_f(k)$ of the golden and faulty model, respectively, indicates a functionally incorrect execution of the *k* instructions.
- **Performance Fault:** When the architectural states match, a difference between the clock cycles $CC_g(k)$ and $CC_f(k)$ of the golden and faulty model, respectively, indicates a functionally correct but performance-impacted execution of the *k* instructions.
- **Latent Fault:** When the architectural states and program execution durations match, a difference between the machine states $MS_g(k)$ and $MS_f(k)$ of the golden and faulty model, respectively, indicates that the fault affected a part of the microprocessor that is not visible to the programmer and did not impact the functional correctness or the performance of executing the *k* instructions. Yet it is not guaranteed that it will not affect future instructions executed on the microprocessor.
- **Masked Fault:** When no discrepancy exists between the architectural states, program execution durations, and machine states of the golden and faulty model, respectively, the fault is suppressed and does not leave any residual effect that may impact future instructions executed on the microprocessor.

Besides being classified in one of the above types, an injected fault may lead to a different simulation outcome, namely stalling of the pipeline. As explained in Section 3, the IVM microprocessor model lacks support for certain instructions, such as system calls and floating-point operations [19]. Even though the window of *k* instructions is carefully chosen so that no such instruction is fetched during program execution on the golden microprocessor model, a fault may still cause the microprocessor to incorrectly call such instructions within the same window. In this case, execution stalls due to the described microprocessor model limitations, preventing classification of the fault in one of the above types. Such faults are reported separately by marking the corresponding runs as *Stalled*.

In the case of *performance faults*, the above method also yields a quantitative assessment of the performance impact by providing the difference in the number of clock cycles for executing the *k* instructions, $CC_f(k) - CC_g(k)$. Interestingly, this difference may some times be negative,

[3] The definitions of architectural state and architectural state corruption used herein are borrowed from [20], 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."

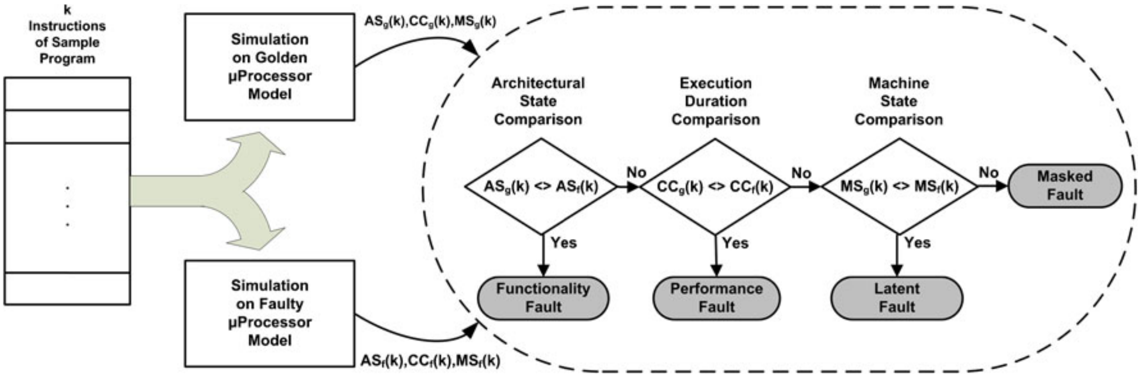


Fig. 4 Simulation outcome classification

i.e. the execution on the faulty microprocessor may be actually faster than the execution on its fault-free counterpart. This is expected, due to the speculative nature of the hardware affected by such performance faults. For example, a fault in the branch prediction unit which results in predicting all branches as "Not Taken", may increase performance when running a program in which most of the branches should, indeed, not be taken, yet the branch prediction unit incorrectly predicts some of them as "Taken". Nevertheless, since speculative hardware is carefully designed to improve overall execution, such oddities are the minority. Most faults in this hardware are expected to adversely impact performance.

As a final note regarding the number and impact of performance faults, as assessed by fault-simulating a representative workload, we raise caution that they only serve as an indication of the magnitude of the problem. Indeed, a considerable number of faults reported as masked may actually be performance faults that have not been activated during the execution of this particular workload. For example, speculative processors use a number of tables to predict the branch target address. Any fault in these tables may cause the running program to jump to an unintended address and, subsequently, flush the pipeline when the real target address is determined. However, only a small subset of such faults will typically be activated during execution of a sample workload, resulting in a reported number of performance faults that is only a lower-bound and, most probably, an underestimate. Therefore, in an effort to provide a more accurate count, we also manually perform a structural analysis of the targeted microprocessor modules, seeking faults akin to the ones that have been classified as performance faults. For example, if a fault in one bit of a register has been shown to be a performance fault, we include in our list all faults in all bits of the register. We call these faults *potential performance faults* and we report them separately. To our knowledge, there currently does not exist any method for either automatically identifying performance faults or automatically generating workload that will be affected by such faults, hence we rely on simulation of typical workload.

6 Results and Analysis

In this section, we discuss the simulation setup used for our study and we present and analyze the results.

6.1 Simulation Setup

Target Modules & Types of Injected Faults In this study, we employ three modules of the IVM microprocessor namely the `ROB`, the `Scheduler` and the `Fetch Unit`. Given the varying degree of performance-enhancing functionality of each of these three modules, which is briefly described in Section 3, we expect to find a significantly different number of performance faults in each of them. The single stuck-at fault model is employed throughout our simulations and faults are injected using the RTL model fault injection technique of [19], which was briefly described in Section 4. The second column of Table 1 reports the total number of faults in each of the three modules. The third column reports the number of faults that cause the pipeline to stall due to limitations in the IVM model, as was explained in the previous section. These faults are disregarded and our study focuses on the remaining faults, which are listed in the fourth column of the table. Among these, a thorough manual analysis of the functionality of each module reveals the number of potential performance faults, which are listed in the fifth column of the table. As expected, the `Fetch Unit`, which encompasses the branch prediction method of IVM, devotes a very large percentage of its real-estate to performance enhancement operations. The `ROB`, which is in charge of ensuring in-order retirement of out-of-order executed instructions, also comprises a considerable number of potential performance faults. Even the `Scheduler`, through its speculative execution functionality, allows for a tangible possibility of performance faults.

Table 1 Statistics on RT-level faults in each module

IVM module name	Total number of faults	Faults causing stalling	Faults considered in study	Potential performance faults
Scheduler	18,822	7,980	10,842	720 (6.64 %)
ROB	61,470	18,802	42,668	16,787 (39.34 %)
Fetch unit	364,490	742	363,748	321,675 (88.43 %)
Total	444,782	27,524	417,258	339,182 (81.28 %)

Simulation Workload In order to investigate the impact of performance faults, we used four quite different SPEC2000 benchmarks as the simulation workload for the IVM processor, namely `gcc`, `bzip2`, `gzip` and `mcf`. The use of four benchmarks ensures variability on the instructions executed through the processor and the control logic that they exercise. Each benchmark is first executed on the golden (fault-free) microprocessor model to obtain the baseline performance and then on each faulty model, in order to apply the performance impact analysis method of Section 5. 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 not implemented in IVM and reaching a code segment that will not stall the pipeline. Then, $k = 10,000$ instructions of each benchmark are executed using the RT-Level Verilog fault simulator. The second column of Table 2 shows the number of clock cycles necessary to execute 10,000 instructions of each benchmark on the golden microprocessor model. The third column lists the number of conditional branches executed within the 10,000 instructions. The fourth and fifth columns list the number of mispredicted branches in the golden model and the average number of mispredicted branches in the faulty models, wherein the faults resulted only in performance degradation, respectively. The difference between these figures accounts, partially, for the performance degradation incurred by such faults, although mispredicted branches constitute only one of the many such reasons.

6.2 Experimental Results

The results of our study are presented and discussed below. Since this is a simulation-based study, the reported numbers should serve as a general indication of the magnitude of the problem rather than an absolute quantification.

Number of Performance Faults First, in Tables 3, 4, 5, and 6, we report the distribution of faults in each of the four fault types discussed in Section 5 for `gcc`, `bzip2`, `gzip` and `mcf`, respectively. For each benchmark, the results are presented individually for each of the three modules and are then accumulated. The first observation that can be made based on the last two columns of these four tables is that only a small subset of faults ends up affecting either the functionality or the performance of executing the 10,000 instructions of each of the four benchmarks. The rest of the faults are either masked or latent. This is expected, since only a small portion of the functionality of the three modules is exercised by the code segments of these benchmarks. As a result, most faults are either not excited at all (or are excited but are logically suppressed), in which case they are reported as masked, or linger around but do not end up impacting the execution of the benchmark within the executed number of instructions, in which case they are reported as latent. We point out that, under appropriate excitation, some of these masked or latent faults will end up causing functional discrepancy, while others will end up causing performance degradation. This can also be corroborated by contrasting the number of identified performance faults to the number of potential performance faults reported in the last column of Table 1.

We now turn our focus to the faults that were classified as either functionality or performance faults during the execution of the 10,000 instructions of the two benchmarks. The distribution of these faults is shown individually for each module and cumulatively in Figs. 5, 6, 7 and 8 for `gcc`, `bzip2`, `gzip` and `mcf`, respectively. Evidently, among the faults that end up affecting the execution of the 10,000 instructions of the four benchmarks, the vast majority only impacts performance but not functionality. Of course, these results are skewed by the fact that we are experimenting with modules whose functionality is closely related to performance enhancement, such as the `Scheduler`, the `ROB`, and the `Fetch Unit`. Nevertheless, *the key takeaway point from the reported data is that there exists a significant number of faults that will only cause performance degradation but no functional discrepancy in the execution of a program.*

Incurred Performance Degradation The second question we try to address concerns the magnitude of performance degradation that the identified performance faults incur. Tables 7, 8, 9 and 10 report the minimum, maximum, and average performance degradation incurred by performance faults in the execution of `gcc`, `bzip2`, `gzip` and

Table 2 Statistics for executing 10,000 instructions of each benchmark

Benchmark	Number of clock cycles to execute 10,000 instructions	Number of conditional branches	Number of mispredicted branches in golden model	Average number of mispredicted branches in models with performance faults
gcc	5,156	1,270	51	107
bzip2	6,127	823	21	45
gzip	16,299	218	13	51
mcf	19,084	357	12	114

Table 3 Fault classification results for gcc

Module name	Total faults	Functionality faults	Performance faults	Latent faults	Masked faults
Scheduler	10,842	149 (1.3 %)	530 (4.8 %)	4,056 (37.4 %)	6,107 (56.3 %)
ROB	42,668	945 (2.2 %)	9,845 (23.0 %)	9,514 (22.3 %)	22,364 (52.4 %)
Fetch unit	363,748	10 (< 0.1 %)	3,420 (0.9 %)	132,074 (36.3 %)	228,244 (62.7 %)
Total	417,258	1,104 (0.3 %)	13,795 (3.3 %)	145,644 (34.9 %)	256,715 (61.5 %)

Table 4 Fault classification results for bzip2

Module name	Total faults	Functionality faults	Performance faults	Latent faults	Masked faults
Scheduler	10,842	92 (1.0 %)	549 (5.0 %)	3,908 (36.0 %)	6,293 (58.0 %)
ROB	42,668	603 (1.4 %)	8,677 (20.3 %)	10,890 (25.5 %)	22,498 (52.7 %)
Fetch unit	363,748	15 (< 0.1 %)	7,284 (2.0 %)	130,305 (35.8 %)	226,144 (62.2 %)
Total	417,258	710 (0.2 %)	16,510 (3.9 %)	145,103 (34.8 %)	254,935 (61.1 %)

Table 5 Fault classification results for gzip

Module name	Total faults	Functionality faults	Performance faults	Latent faults	Masked faults
Scheduler	10,842	41 (0.4 %)	215 (2.0 %)	4,051 (37.4 %)	6,535 (60.2 %)
ROB	42,668	423 (1.0 %)	4,416 (10.3 %)	14,022 (32.9 %)	23,807 (55.8 %)
Fetch unit	363,748	10 (< 0.1 %)	1,279 (0.3 %)	133,045 (36.6 %)	229,414 (63.0 %)
Total	417,258	474 (0.1 %)	5,910 (1.4 %)	151,118 (36.2 %)	259,756 (62.2 %)

Table 6 Fault classification results for mcf

Module name	Total faults	Functionality faults	Performance faults	Latent faults	Masked faults
Scheduler	10,842	9 (< 0.1 %)	16 (0.1 %)	3,922 (36.2 %)	6,895 (63.6 %)
ROB	42,668	257 (0.6 %)	2,192 (5.1 %)	16,219 (38.0 %)	24,000 (56.2 %)
Fetch unit	363,748	7 (< 0.1 %)	1,109 (0.3 %)	133,235 (36.7 %)	229,397 (63.0 %)
Total	417,258	273 (< 0.1 %)	3,317 (0.8 %)	153,376 (36.7 %)	260,292 (62.4 %)

Fig. 5 Distribution of functionality and performance faults for gcc

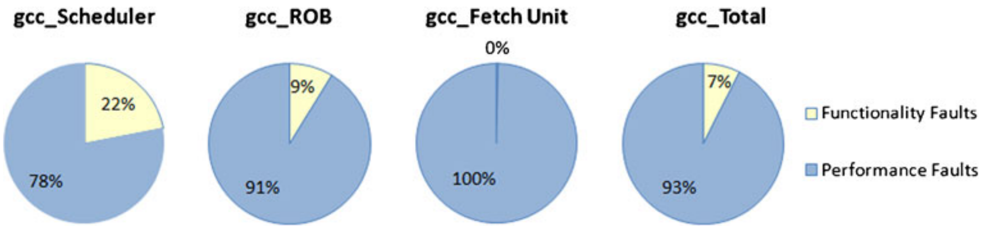


Fig. 6 Distribution of functionality and performance faults for bzip2

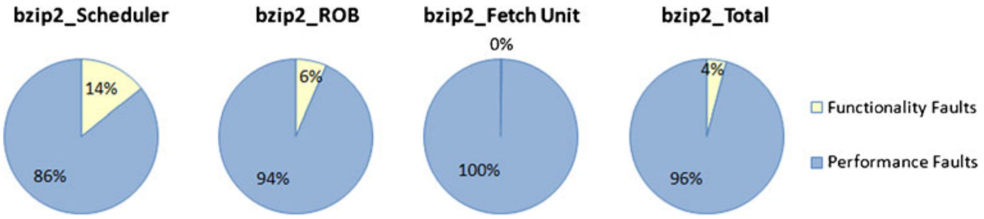


Fig. 7 Distribution of functionality and performance faults for gzip

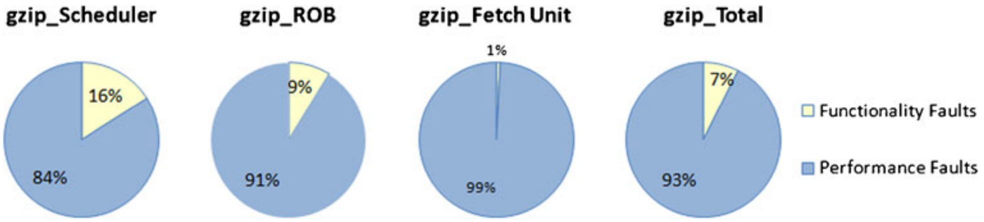


Fig. 8 Distribution of functionality and performance faults for mcf

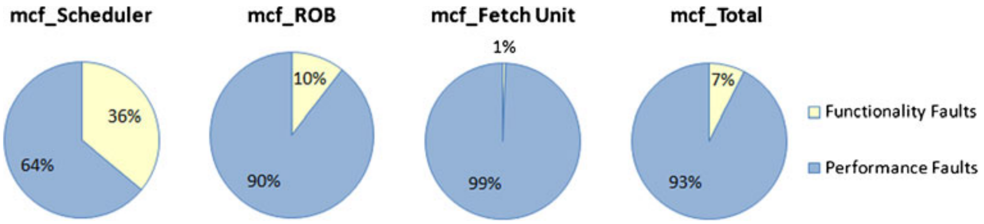


Table 7 Performance degradation incurred in gcc (Baseline = 5, 156 clock cycles)

Module	Minimum	Maximum	Average
Scheduler	−38(−0.7 %)	+4, 621(+89.6 %)	+569(+11.0 %)
ROB	−13(−0.2 %)	+88, 694(+1720.0 %)	+778(+15.0 %)
Fetch unit	−249(−4.8 %)	+16, 990(+329 %)	+2, 213(+42.9 %)
Overall	−249(−4.8 %)	+88, 694(+1720.0 %)	+1, 126(+21.8 %)

Table 9 Performance degradation incurred in gzip (Baseline = 16, 299 clock cycles)

Module	Minimum	Maximum	Average
Scheduler	−2, 219(−13.6 %)	+1, 811(+11.1 %)	+122(+0.7 %)
ROB	−867(−5.3 %)	+72, 686(+445 %)	+603(+3.7 %)
Fetch unit	−776(−4.8 %)	+1, 337(+8.2 %)	+144(+0.9 %)
Overall	−2, 219(−13.6 %)	+72, 686(+445 %)	+486(+2.9 %)

mcf, respectively. The provided figures show the difference in the number of clock cycles that it takes to complete the 10,000 instructions in the presence of a performance fault. We remind that the number of clock cycles it takes to execute these instructions in the golden model is 5,156 for gcc, 6,127 for bzip2, 16,299 for gzip and 19,084 for mcf.

As can be observed, due to the branch prediction and other speculative execution aspects of modern microprocessors, some performance faults actually speed up the execution of the instructions. Hence, the minimum performance degradation is negative, i.e. it is a performance improvement. At the other end of the spectrum, the worst performance faults incur a very large degradation, often orders of magnitude worse that the performance of the golden model. On average, the identified performance faults incur a performance degradation of 1,126 clock cycles (21.8 %) in the execution of gcc, 779 clock cycles (12.7 %) in the execution of bzip2, 486 clock cycles (2.9 %) in the execution of gzip and 1,193 clock cycles (6.2 %) in the execution of mcf. *The key takeaway point from the reported data is that the impact of performance faults is quite significant, warranting further investigation of methods for recovering the lost performance.*

Consistency of Relative Impact The third point that we investigate concerns the relative impact of performance faults on different benchmarks. Specifically, we first examine the number of performance faults that are activated in more than one of the four benchmarks. For example, let us consider the performance faults that occur during he simulation runs of both gcc and bzip2. The results are shown in Fig. 9 individually for each module and cumulatively. Overall, while only 20,013 out of the 339,182 possible performance faults (5.9 %) are activated when executing 10,000 instructions of either gcc or bzip2, which in a uniform distribution would imply a very small intersection set, more than half of them (10,292, i.e. 51.42 %) are actually activated in both benchmarks. The situation is similar for any other pair of benchmarks. To further demonstrate this point, Fig. 10 shows the number of performance faults that are activated in at least one, two, three, or all four benchmarks, individually for each module as well as cumulatively. *These results imply that some performance faults have consistently a much higher probability of activation, independent of the workload being executed by the processor, hence they are more critical.*

By further examining the impact of performance faults that are activated in more than one benchmark we obtain another very interesting result that corroborates our observation regarding their relative importance. Consider, for example, the performance faults that are activated in both of the two benchmarks gcc and bzip2. For each of these two benchmarks, we create a rank-ordered list of all these faults based on decreasing performance degradation impact. If a fault lies in position i on the gcc list and position j on the bzip2 list, we compute $|i - j|$ and we report the average over all faults, individually for each module and cumulatively, in the third column of Table 11. As may be observed, while the overall fault list includes 10,292 faults, the average difference in the ranking of importance of a fault to the two benchmarks is only 198 positions, i.e. 1.92 %, clearly indicating consistency of relative impact of a performance fault across different workloads. Furthermore, we

Table 8 Performance degradation incurred in bzip2 (Baseline = 6, 127 clock cycles)

Module	Minimum	Maximum	Average
Scheduler	−201(−3.2 %)	+1, 668(+27.2 %)	+46(+0.7 %)
ROB	−74(−1.2 %)	+86, 465(+1411.0 %)	+253(+4.1 %)
Fetch unit	−205(−3.3 %)	+19, 253(+314.0 %)	+1, 461(+23.8 %)
Overall	−205(−3.3 %)	+86, 465(+1411 %)	+779(+12.7 %)

Table 10 Performance degradation incurred in mcf (Baseline = 19, 084 clock cycles)

Module	Minimum	Maximum	Average
Scheduler	−1, 872(−9.8 %)	+3, 490(+18.3 %)	+345(+1.8 %)
ROB	−1, 343(−7.0 %)	+83, 008(+435 %)	+1, 079(+5.6 %)
Fetch unit	−520(−2.7 %)	+4, 083(+21.4 %)	+1, 433(+7.5 %)
Overall	−1, 872(−9.8 %)	+83, 008(+435 %)	+1, 193(+6.2 %)

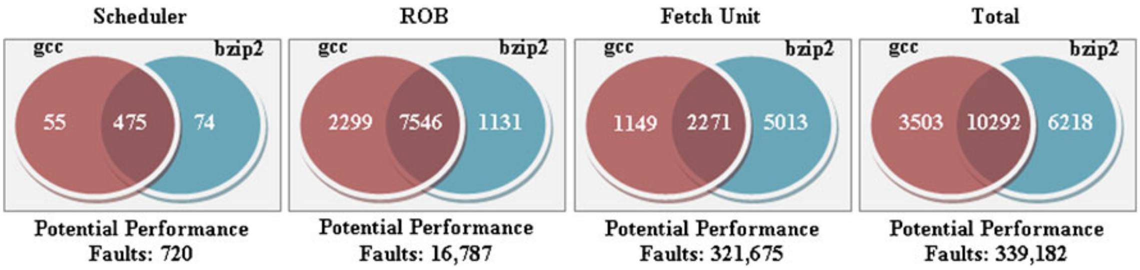


Fig. 9 Consistency of Activated Performance Faults in gcc and bzip2

compare the actual degradation (i.e. additional clock cycles) incurred by each performance fault on the execution of the two benchmarks. Specifically, if a fault incurs x clock cycles of performance degradation on `gcc` and y on `bzip2`, we compute $|x - y|$ and we report the average over all faults, individually for each module and cumulatively, in the fourth column of Table 11. The percentages in the parentheses next to these numbers express the average difference between the impact of faults as a percentage of the average execution time of these two benchmarks (i.e. divided by (5,156+6,127)/2). As may be observed, the average difference in performance degradation incurred by the 10,292 faults that are activated both in `gcc` and `bzip2` in only 722 clock cycles (12.80 %), further supporting our observation regarding consistency of relative impact. It is also worth noting that the top-10 performance faults in the impact-based, rank-ordered lists for `gcc` and `bzip2` are exactly the same. Similar results are obtained for any other pair of benchmarks.

To further demonstrate this point, in Table 12 we report the same information across all four benchmarks. The second column of the table indicates the number of performance faults that affect all four benchmark per module and cumulatively. Once again, we create a rank-ordered list of all these faults based on decreasing performance degradation impact. If a fault lies in position i on the `gcc` list, position j on the `bzip2` list, k on the `gzip` list, and position l on the `mcf` list, we compute $(|i-j|+|i-k|+|i-l|+|j-k|+|j-l|+|k-l|)/6$ and we report the average over all faults in the third column of the table. As may be observed, while the overall list of common performance faults across the four benchmarks has 1,828 elements, the average difference in the rank-ordered lists of the four benchmarks is only 60 positions, i.e. 3.28 %. Similarly, in the fourth column

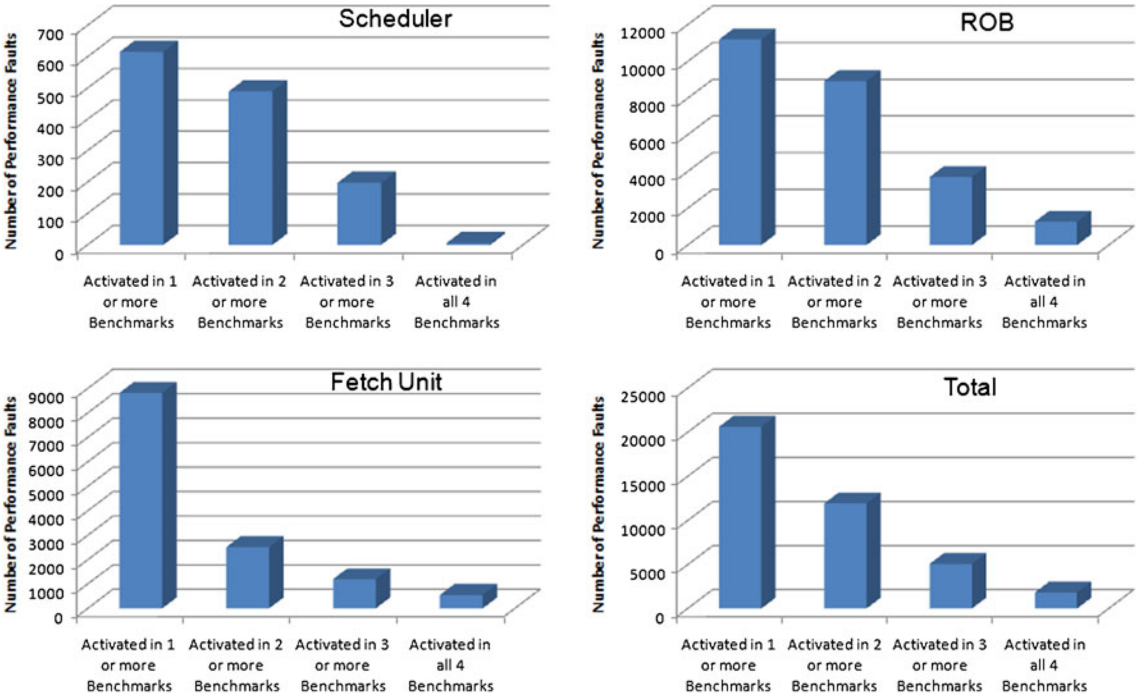


Fig. 10 Commonality of activated performance faults across multiple benchmarks

Table 11 Impact consistency across `gcc` & `bzip2`

Module name	Faults in intersection of gcc & bzip2	Average ranking Difference	Average impact Difference
Scheduler	475	37 (7.78 %)	94 (1.67 %)
ROB	7,546	173 (2.29 %)	605 (10.72 %)
Fetch unit	2,271	74 (3.25 %)	1,240 (21.97 %)
Overall	10,292	198 (1.92 %)	722 (12.80 %)

we compare the actual degradation that a performance fault causes in each of the four benchmarks. Specifically, if a fault incurs x clock cycles of performance degradation on `gcc`, y on `bzip2`, z on `bzip2` and w on `bzip2`, we compute $(|x-y|+|x-z|+|x-w|+|y-z|+|y-w|+z-w|)/6$ and we report the average over all faults, individually for each module and cumulatively. The percentages in the parentheses next to these numbers express the average difference between the impact of faults as a percentage of the average execution time of the four benchmarks (i.e. divided by (5,156+6,127+16,299+19,084)/4). The results show that the average difference in performance degradation incurred by the 1,828 faults in the intersection of the four benchmarks is only 1,606 clock cycles (13.77 %), further corroborating our observation regarding impact consistency of performance faults.

Based on the above observations, *the key takeaway conjecture is that the activation probability, as well as the relative impact of performance faults are consistent across different benchmarks. Hence, additional hardware expended towards alleviating the impact of such faults and reclaiming the lost performance would benefit the entire microprocessor workload.*

7 Examples of Performance Faults

In this section, we discuss in detail and we experimentally demonstrate the impact of a few examples of performance faults in IVM using SPEC2000 benchmarks. We note that the impact of each such fault is reflected in a difference between the number of clock cycles needed to execute a specific number of instructions on the faulty and golden microprocessor model. We note, however, that this difference may be either positive or negative, implying a performance degradation or a performance improvement, respectively. The results are summarized in Table 13, where each row corresponds to one of these examples. The second, third, and fourth column of the table list the running benchmark, the name of the IVM signal on which a fault is injected, and the number of executed instructions, respectively. The last two columns provide the number of the clock cycles required to run these instructions in the two models.

Table 12 Impact consistency across all four benchmarks

Module name	Faults in intersection of all Benchmarks	Average ranking Difference	Average impact Difference
Scheduler	6	0 (0.00 %)	1,853 (15.88 %)
ROB	1,272	26 (2.04 %)	643 (5.51 %)
Fetch Unit	550	44 (8.00 %)	3,832 (32.85 %)
Overall	1,828	60 (3.28 %)	1,606 (13.77 %)

Example 1 As was explained in Section 3.1, up to 6 instructions may be issued by IVM in each clock cycle, including up to 2 simple, 1 complex, 1 branch and 2 memory instructions. For each functional unit, in turn, the Scheduler examines the array of available instructions, picks the first one that matches the type of the functional unit and places it to the corresponding output port. At the same time, it sets the *valid* bit of this output port to ‘1’, in order to notify the functional unit that it can commence execution. Only after this *valid* bit is set to ‘1’ is the *issued* bit corresponding to this instruction also set to ‘1’, signifying that the instruction has been issued.

Let us assume now that, in the golden model, a simple instruction i is issued for execution to $simple\_unit\_1$ in clock cycle c by placing it to the corresponding output port and setting the $valid\_simple1$ signal to ‘1’. If a stuck-at 0 fault is injected in *valid_simple1* signal, then $simple\_unit\_1$ will not start executing instruction i and the *issued* signal corresponding to this instruction will remain ‘0’. Immediately thereafter, the Scheduler will seek an available instruction to issue for execution to $simple\_unit\_2$ in clock cycle c. The first such unissued instruction will again be instruction i. Hence, the instruction that would be issued to $simple\_unit\_2$ in clock cycle c if the $valid\_simple1$ signal was not stuck at ‘0’ will now have to wait since there is no other simple unit available in clock cycle c. The same process will be repeated every time the Scheduler will try to issue an instruction to $simple\_unit\_1$, effectively making the processor operate with only one simple unit. Instructions will still be executed correctly, yet the throughput will be lower. Hence, the fault $valid\_simple1$ stuck-at 0 is a performance fault. As shown in the first row of Table 13, when running 50,000 instructions of `gcc`, this fault incurs a performance degradation of 15.42 %.

Example 2 A similar situation occurs with the two memory units in IVM: a stuck-at 0 fault in $valid\_memory2$ will effectively make IVM operate with only one memory unit and will prolong the run time of programs. However, in some odd cases this fault may actually lead to improved

Table 13 Impact of example performance faults

Ex.	Work-load	Target signal	# of Instr.	# of Clock cycles (golden)	# of Clock cycles (faulty)
1	gcc	valid_simple1	50,000	41,724	48,162
2	gcc	valid_memory2	10,000	5,156	5,442
2	gcc	valid_memory2	50,000	41,724	41,256
3	bzip2	rob_pctag_f	15,000	8,464	10,530
3	bzip2	rob_pctag_f	50,000	22,748	22,654
4	bzip2	tk_addr	50,000	22,748	67,015

performance. This happens because when all instructions end up using the same memory unit, more internal data forwarding can be performed when load aliasing occurs, thus speeding up execution. The second and third rows of Table 13 quantify the impact of this performance fault when executing 10,000 and 50,000 instructions of `gcc`, respectively. In the first case, a performance degradation of 5.54 % is incurred, while in the second case, performance actually improves by 1.12 %!

Example 3 In IVM, in order to enforce in-order commitment of the out-of-order executed instructions, each of the 64 ROB entries holding an instruction also includes a 64-bit field rob_pctag_f, which holds the PC of the next instruction. Thus, when a branch instruction is ready to be retired, its corresponding entry in the ROB contains both the correct target PC address and the actual PC address that the next instruction was fetched from based on the predicted branch direction. Prior to retiring a branch instruction, these two fields are compared and, if they do not match, the pipeline is flushed and the correct target PC address of the branch is sent to the Fetch Unit to resume execution accordingly.

Assume now that a branch is correctly predicted, but due to a stuck-at fault in the field rob_pctag_f, the above comparison fails, causing the ROB to assume that the branch prediction was incorrect and to flush the pipeline, thus adversely affecting performance. The next time around, the branch will most likely end up in a different entry of the ROB and the comparison will succeed, so the workload will still be executed correctly. We should also mention that, as discussed in Section 3.3, whenever a branch instruction retires, the ROB sends the corresponding branch direction (i.e. Taken or Not Taken) to the Fetch Unit, which uses it along with other parameters to decide the prediction of subsequent branches. Thus any unexpected branch flush, such as the one described above, may alter the subsequent branch predictions and, in turn, the stream of fetched instructions. Interestingly, since predictions are not always correct, this may lead to either performance degradation or performance gain. The fourth and fifth rows of Table 13 quantify the impact of two different stuck-at faults in a rob_pctag_f entry while executing 15,000 and 50,000 instructions of `bzip2`, respectively. In the first case, a performance degradation of 24.40 % is incurred, while in the second case, performance actually improves by 0.41 %!

Example 4 When the Fetch Unit of IVM receives a branch instruction, it employs a multi-level method to predict whether the branch should be taken or not, as well as to determine the branch target address in the case that it is predicted as taken. Suppose that the register tk_addr, which holds the target address of given branch instruction has a stuck-at fault. In this case, if this branch is predicted as taken, the subsequent instructions are fetched from an incorrect address. Eventually, however, even if the branch direction was correctly predicted as taken, the ROB will identify the discrepancy between the predicted branch target address and the correct one, as explained in the previous example. Hence, the pipeline will be flushed and the operation will resume from the correct branch target address. Nevertheless, resources and clock cycles will have been wasted on unnecessary instructions, effectively degrading performance. The last row in Table 13 quantifies the impact of a stuck-at fault in tk_addr when executing 50,000 instructions of `bzip2`, which degrades by 194.59 %.

8 Utility of Performance Impact Info

Information regarding the relative impact of performance faults may be utilized to guide design modifications for recovering the lost performance. In other words, akin to the traditional fault-tolerant design approach, which ensures correct functionality in the presence of faults, one may think of this as a performance degradation-tolerant design approach, which ensures the expected performance level. Consider, for example, the IVM branch prediction method, as discussed in Section 3.3. The process involves a local predictor with 1024 three-bit locations and a gshare predictor with 4096 two-bit locations. Faults in these tables do not affect functionality but only impact performance. Nevertheless, equipping these tables with error recovery mechanisms ensures their correct operation and, thereby, recovers the lost performance. As a point of reference, in Example 4 of Section 7, 2,356 out of the 4,117 (57.22 %) branch instructions were mispredicted in the faulty model, as compared to only 21 branch instructions (0.51 %) that were mispredicted in the golden model. Use of memory error recovery methods could potentially recover the performance loss.

In addition to hardware modifications for ensuring performance, one may also envision the use of additional hardware to convert actual functionality faults into performance faults, thereby improving yield. In other words, through such hardware, a device that in the presence of a fault would be discarded as faulty, may be salvaged since it can operate correctly, yet at reduced performance. For example, as we discussed in Section 3.2, up to 8 instructions are retired from the ROB module in the IVM processor in each clock cycle. A fault in any one of the 8 ROB output ports will result in a functionality fault, since the first instruction to be stored there will never retire, stalling the pipeline. A small self-test controller examining the operational health of these ports and isolating the faulty one would enable the processor to continue operating, yet with degraded performance. With this solution, execution of 40,000 instructions of gcc, which take 33,024 clock cycles in the golden model and which would never execute in the faulty model, may now be executed in 33,925 clock cycles, i.e. with a performance degradation of 2.72 %.

9 Conclusion

Various architectural features aiming to improve microprocessor performance give rise to a new type of faults which do not affect correctness but only prolong program execution. As we demonstrated through a quantitative study employing the IVM microprocessor model and SPEC2000 benchmarks, a sizeable number of faults in various modules of a microprocessor are, indeed, such performance faults and the incurred performance degradation is, often, very significant. Interestingly, the relative impact of performance faults is consistent across different workloads. Hence, besides extending our study to more modules and benchmarks, the continuation of this research will also investigate methods for proving that a fault can only cause performance degradation and automatically generating appropriate test sequences, as well as hardware methods for recovering the incurred performance loss and improving yield. Finally, an additional application of the identification of performance faults is in the development of compact Software-Based Self Test (SBST) methods. In such methods, marking performance faults prior to the test generation process, could extensively decrease the number of test patterns applied during each test as well as the test application time.

Acknowledgments This work is supported by a generous gift from Intel Corp. The first author performed this research while being a visiting student at Yale University. A preliminary version of parts of the work reported herein was presented at the 2009 International Conference on Computer Design [27]. The authors would like to thank Prof. Sanjay Patel and Nicholas Wang from the University of Illinois at Urbana-Champaign for sharing the IVM microprocessor model and for providing technical assistance in its installation and usage.

References

1. Jouppi NP (1990) Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In: International symposium on computer architecture, pp 364–373
2. Kroft D (1981) Lookup-free instruction fetch/prefetch cache organization. In: International symposium on computer architecture, pp 81–87
3. McFarling S (1993) Combining branch predictors, Tech. Rep. TR TN-36
4. Sprangle E, Chappell R, Alsup Ms, Patt Y (1997) The agree predictor: a mechanism for reducing negative branch history interference. In: International symposium on computer architecture, pp 284–291
5. Wang K, Franklin M (1997) Highly accurate data value prediction using hybrid predictors. In: International symposium on microarchitecture, pp 281–290
6. Yeh TY, Patt YN (1992) Alternative implementations of two-level adaptive branch prediction. In: International symposium on computer architecture, pp 124–134
7. Loh GH (2006) Revisiting the performance impact of branch predictor latencies. In: International symposium on performance analysis of systems and software. IEEE Computer Society, Los Alamitos, pp 59–69
8. Hao E, Chang PY, Patt YN (1994) The effect of speculatively updating branch history on branch prediction accuracy, revisited. In: International symposium on microarchitecture, pp 228–232
9. Jimenez DA (2002) Delay-sensitive branch predictors for future technologies, Ph.D. thesis, University Texas at Austin
10. Almukhaizim S, Verdel T, Makris Y (2003) Cost-effective graceful degradation in speculative processor subsystems: the branch prediction case. In: International conference on computer design. IEEE Computer Society, Los Alamitos, pp 194–197
11. Almukhaizim S, Petrov P, Orailoglu A (2001) Low-cost, software-based self-test methodologies for performance faults in processor control subsystems. In: Custom integrated circuits conference, pp 263–266
12. Almukhaizim S, Petrov P, Orailoglu A (2001) Faults in processor control subsystems: testing correctness and performance faults in the data prefetching unit. In: Asian test symposium. IEEE Computer Society, Los Alamitos, pp 319–324
13. Gizopoulos D, Paschalis A, Zorian Y (2004) Embedded processor-based self-test. Kluwer
14. Hatzimihail M, Psarakis M, Gizopoulos D, Paschalis A (2007) A methodology for detecting performance faults in microprocessors via performance monitoring hardware. In: International test conference, pp 1–10
15. Sanchez E, Reorda MS, Tonda (2011) On the functional test of branch prediction units based on branch history table. In: International conference on VLSI and system-on-chip, pp 278–283
16. Bernardi P, Ciganda L, Grosso M, Sanchez E, Sonza Reorda M (2012) A SBST strategy to test microprocessors' branch target buffer. In: International symposium on design and diagnostics of electronic circuits & systems, pp 306–311
17. Sato T (1998) First step to combining control and data speculation. In: International workshop on innovative architecture, pp 53–60
18. Gonzalez J, Gonzalez A (1998) The potential of data value speculation to boost ILP. In: International conference on supercomputing, pp 21–28
19. Karimi N, Maniatakos M, Makris Y, Jas A (2008) On the correlation between controller faults and instruction-level errors

in modern microprocessors. In: International test conference, pp 24.1.1–24.1.10

20. Wang NJ, Quek J, Rafacz TM, Patel SJ (2004) Characterizing the effects of transient faults on a high-performance processor pipeline. In: International conference on dependable systems and networks, pp 61–70
21. Wang NJ, Mahesri A, Patel SJ (2007) Examining ACE analysis reliability estimates using fault-injection. SIGARCH Comput Archit News 35(2):460–469
22. Burger D, Austin TM, Bennett S (1996) Evaluating future microprocessors: the simplescalar tool set, Tech. Rep CS-TR-1996-1308
23. Hennessy JL, Patterson DA (2003) Computer architecture: a quantitative approach, 3rd edn. Morgan Kaufmann, San Francisco, CA
24. Wang NJ, Patel SJ (2005) Restore: symptom based soft error detection in microprocessors. In: International conference on dependable systems and networks, pp 30–39
25. Maniatakos M, Karimi N, Tirumurti C, Jas A, Makris Y (2009) Instruction-level impact comparison of RT- vs. gate-level faults in a modern microprocessor controller. In: VLSI test symposium, pp 9–14
26. Baraza JC, Gracia J, Blanc S, Gil D, Gil PJ (2008) Enhancement of fault injection techniques based on the modification of VHDL code. IEEE Trans VLSI 16(6):693–706
27. Karimi N, Maniatakos M, Tirumurti C, Jas A, Makris Y (2009) Impact analysis of performance faults in modern microprocessors. In: International conference on computer design, pp 91–96

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 master's 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 worked as a postdoctoral researcher at Duke University for one year and she is currently a visiting assistant professor at Polytechnic Institute of New York University, NYU-Poly. Her research interests include design-for testability, concurrent testing, fault tolerance, reliability enhancement, and hardware security.

Michail Maniatakos received the BS and MS degrees in computer science and embedded systems from the University of Piraeus, Greece, in 2006 and 2007, respectively, as well as the MS and Ph.D. degrees in electrical engineering from Yale University, New Haven, Connecticut, in 2008 and 2012, respectively. He is currently, an Assistant Professor of Electrical and Computer Engineering at New York University in Abu Dhabi, UAE. His research interests include test and reliability of modern microprocessors and computer architecture.

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.

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 he is the program chair of the 2013 IEEE VLSI Test Symposium and the 2012 Test Technology Education Program (TTEP) of the IEEE Test Technology Technical Council (TTTC).