Impact Analysis of Performance Faults in Modern Microprocessors

Naghmeh Karimi, *ECE Department*, *University of Tehran*
Michail Maniatakos, *EE Department*, *Yale University*
Chandra Tirumurti, *Strategic CAD Labs*, *Intel Corporation*
Abhijit Jas, *Validation and Test Solutions*, *Intel Corporation*
Yiorgos Makris, *EE Department*, *Yale University*

Abstract— **Towards improving performance, 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. 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 and we discuss how this information may guide the inclusion of additional hardware for performance loss recovery and yield enhancement.**

I. Introduction

In their constant 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 prefetching, data value and branch prediction are only a few such techniques to be mentioned [1], [2], [3], [4], [5], [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], [8], [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], [11], [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 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 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 to performance faults.

The rest of this paper is organized as follows. In section II, we take a closer look at the concept of performance faults and the underlying architectural features that facilitate their existence. Then, in section III, we describe the microprocessor model which serves as a vehicle for this study. In section IV, we discuss the capabilities of the simulation infrastructure which is used in this study. The employed performance impact analysis method is detailed in section V and extensive results quantifying the impact of performance faults are presented in section VI. In section VII, we discuss the potential utility of this study in guiding hardware addition for performance loss recovery and yield enhancement. Conclusions are drawn in section VIII.

II. Performance Faults

Among the 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, then, freed for use by other instructions, increasing the overall performance. Faults preventing 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 [13], [14]. 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 [15]. 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!

III. Study Framework

Our investigation on the impact of performance faults builds upon a previously developed fault simulation infrastructure, which is presented in detail in [16]. The employed model is the Verilog implementation of an Alpha-like microprocessor, called IVM (Illinois Verilog Model) [17], [18]. 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 part of the SimpleScalar tool suite and supports the full instruction set of the Alpha 21264 microprocessor [19]. 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.

In this research, we focus on three key control modules of the IVM microprocessor, namely the Scheduler, the ReOrder Buffer (ROB), and the Fetch Unit.

IV. 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[1] which was developed and presented in detail in [16], wherein fault injection is performed using a method similar to the parallel saboteurs technique described in [21]. Specifically, the Verilog model of each target module is mutated and a

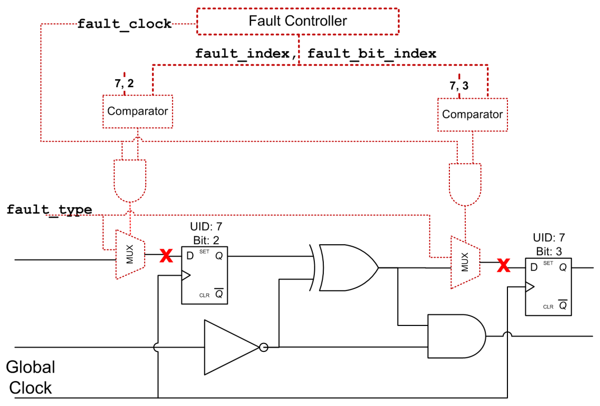

Fig. 1. RT-Level Fault Injection Method

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. 1 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 storage element or wire defined in the RT-Level Verilog model (dotted lines indicate hardware added for fault simulation). As shown, each element (wire or storage element) 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.

V. Performance Impact Analysis Methodology

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. 2. Specifically, we start by simulating the execution of k instructions of a typical workload and recording the corresponding number of clock cycles, $CC_g(k)$, along with the resulting architectural state, $AS_g(k)$, 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

[1]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 [20]; hence, we expect that the obtained results are representative of what would be obtained through gate-level fault simulation.

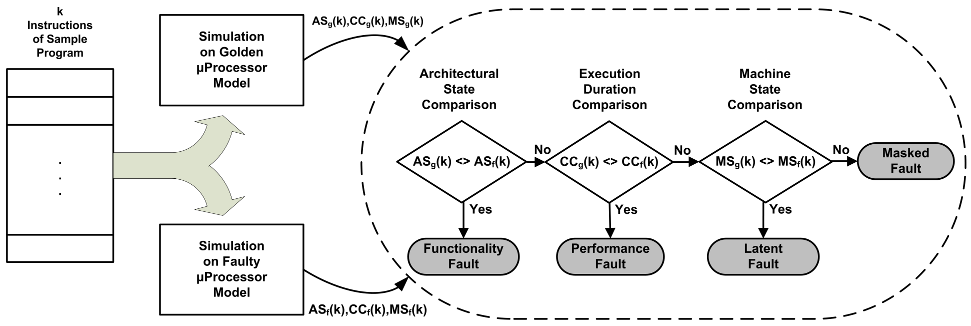

Fig. 2. Simulation Outcome Classification

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 III, the IVM microprocessor model lacks support for certain instructions, such as system calls and floating-point operations [16]. 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, 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 perform a structural analysis of the targeted microprocessor modules, seeking faults akin to the ones that have been classified as performance faults.

VI. Results and Analysis

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

A. 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, 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 RT-Level model fault injection technique of [16], which was briefly described in section IV. The second column of Table I 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 faults that could potentially be 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 large percentage of its real-estate to performance enhancement. The `ROB`, which is in charge of ensuring in-order retirement of out-of-order executed instructions, also comprises a considerable number of performance faults. Even the `Scheduler`, through its speculative execution functionality, allows for a tangible possibility of such performance faults.

Simulation Workload: In order to investigate the impact of performance faults, we used two quite different SPEC2000 benchmarks as the simulation workload for the IVM processor, namely `gcc` and `bzip2`. The use of two 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 V. 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 number of clock cycles necessary to execute 10,000 instructions on the golden microprocessor model is 5,156 for `gcc` and 6,127 for `bzip2`.

Computational Power: The reported simulations were performed over the course of four months on two Quad-core Xeon 3.33GHz servers with 16GB of memory.

B. 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 II and III, we report the distribution of faults in each of the four fault types discussed in section V for `gcc` and `bzip2`, 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 two tables is that only a small subset of faults (approx. 4%) ends up affecting either the functionality or the performance of executing the 10,000 instructions of each of the two benchmarks. The rest of the faults are either masked (approx. 61%) or latent (approx. 35%). 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 I.

TABLE I
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	7,891	10,931	720 (6.58%)
ROB	61,470	18,027	43,443	16,787 (38.64%)
Fetch Unit	364,490	731	363,759	321,675 (88.43%)
Total	444,782	26,649	418,133	339,182 (81.11%)

As may be observed in these results, among the faults that are classified as either functionality or performance faults during the execution of the 10,000 instructions of the two 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 IV and V report the minimum, maximum, and average performance degradation incurred by performance faults in the execution of `gcc` and `bzip2`, 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` and 6,127 for `bzip2`.

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 mag-

TABLE II
FAULT CLASSIFICATION RESULTS FOR GCC

Module Name	Total Faults	Functionality Faults	Performance Faults	Latent Faults	Masked Faults
Scheduler	10,931	113 (1.0%)	497 (4.5%)	3,788 (34.6%)	6,533 (59.7%)
ROB	43,443	1,261 (2.9%)	9,559 (22.0%)	9,810 (22.5%)	22,813 (52.1%)
Fetch Unit	363,759	9 (<0.1%)	3,425 (1.0%)	132,083 (36.3%)	228,242 (62.7%)
Total	418,133	1,383 (0.3%)	13,481 (3.2%)	145,681 (34.8%)	257,588 (61.6%)

TABLE III
FAULT CLASSIFICATION RESULTS FOR BZIP2

Module Name	Total Faults	Functionality Faults	Performance Faults	Latent Faults	Masked Faults
Scheduler	10,931	112 (1.0%)	555 (5.0%)	3,926 (35.9%)	6,338 (57.9%)
ROB	43,443	852 (1.9%)	8,737 (20.1%)	11,098 (25.5%)	22,756 (52.3%)
Fetch Unit	363,759	16 (<0.1%)	7,293 (2.0%)	130,306 (35.8%)	226,144 (62.1%)
Total	418,133	980 (0.2%)	16,585 (3.9%)	145,330 (34.7%)	255,238 (61.0%)

TABLE IV
PERFORMANCE DEGRADATION INCURRED IN GCC (BASELINE = 5,156 CLOCK CYCLES)

Module	Minimum	Maximum	Average
Scheduler	-38 (-0.7%)	+4,621 (+89.6%)	+570 (+11%)
ROB	-13 (-0.2%)	+88,694 (+1720%)	+881 (+17%)
Fetch Unit	-249 (-4.8%)	+16,990 (+329%)	+2,087 (+40.4%)
Overall	-249 (-4.8%)	+88,694 (+1720%)	+1,189 (+23%)

TABLE V
PERFORMANCE DEGRADATION INCURRED IN BZIP2 (BASELINE = 6,127 CLOCK CYCLES)

Module	Minimum	Maximum	Average
Scheduler	-201 (-3%)	+1,668 (+27.2%)	+47 (+0.7%)
ROB	-96 (-1.5%)	+86,465 (+1411%)	+273 (+4.4%)
Fetch Unit	-205 (-3%)	+19,253 (+314%)	+1,463 (+23.8%)
Overall	-205 (-3%)	+86,465 (+1411%)	+787 (+12.8%)

nitude worse that the performance of the golden model. On average, the identified performance faults incur a 23% performance degradation in the execution of `gcc` and 13% in the execution of `bzip2`. *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 the simulation runs of both `gcc` and `bzip2`. The results are shown in Fig. 3 individually for each module and cumulatively. Overall, while only 19,959 out of the 339,182 possible performance faults (5.8%) 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 (10,107, i.e. 50.63%) are actually activated in both benchmarks. This clearly implies that some 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 performance faults that are activated in both benchmarks, we obtain another very interesting result that corroborates our observation regarding the relative importance of performance faults. Specifically, for each of the two benchmarks, we create a rank-ordered list of all 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 VI. As may be observed, while the overall fault list includes 10,107 faults, the average difference in the ranking of importance of a fault to the two benchmarks is only 197 positions, i.e. 1.94%, clearly indicating consistency of relative impact of a performance fault across different workloads. Furthermore, we compare the actual degradation (i.e. percentage of additional clock cycles) incurred by each performance fault on the execution of the two benchmarks. Specifically, if a fault incurs $x\%$ 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 VI. As may be observed, the average difference degradation incurred by the 10,107 faults that are activated both in `gcc` and `bzip2` in only 13.86 percentile units, further supporting our observation regarding consistency of relative impact. Finally, 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.

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

VII. UTILITY OF PERFORMANCE IMPACT INFORMATION

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 fault-tolerant design approaches, which ensure correct functionality in the presence of faults, one may think of this as a performance degradation-tolerant design approach, which en-

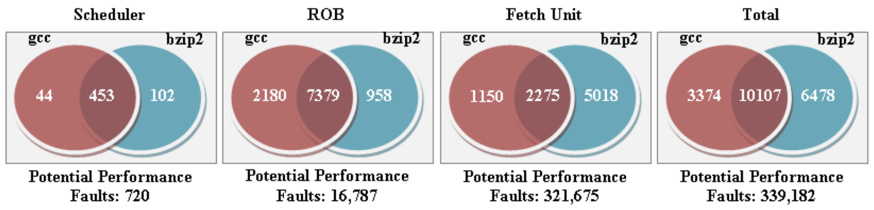

Fig. 3. Consistency of Activated Performance Faults in gcc and bzip2

sures the expected performance level. Consider, for example, the IVM branch prediction method, which 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 do impact performance. Nevertheless, Error Correction Codes (ECC) could be added to these tables to ensure their correct operation and, thereby, recover the lost performance. As a point of reference, when executing 50,000 instructions of `bzip2` on the golden IVM model, only 21 branches (0.51%) are mispredicted; in contrast, when a fault is injected in the branch target address of an instruction, 2,356 out of the 4,117 (57.22%) branches are mispredicted, incurring a performance loss of 194.59%. Use of ECC codes could potentially recover this performance loss.

In addition to hardware modifications for ensuring performance, one may also envision the use of additional hardware to convert functionality faults into performance faults, thereby improving yield. 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, up to 8 instructions may be retired from the ROB module of IVM 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 at degraded performance. With this solution, execution of 40,000 instructions of `gcc`, which take 33,024 clock cycles in the golden model and which do not execute in the faulty model, may now be executed in 33,925 clock cycles, i.e. with a performance degradation of 2.72%.

VIII. 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 hardware methods for recovering the incurred performance loss and improving yield.

TABLE VI
Performance Fault Impact Consistency

Module Name Name	Faults in Intersection of gcc & bzip2	Average Ranking Difference	Average Impact Difference
Scheduler	453	37 (8.16%)	1.89%
ROB	7,379	152 (2.05%)	12.32%
Fetch Unit	2,275	74 (3.25%)	21.25%
Overall	10,107	197 (1.94%)	13.86%

References

[1] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *ISCA*, pp. 364–373, 1990.
[2] D. Kroft, "Lookup-free instruction fetch/prefetch cache organization," *ISCA*, pp. 81–87, 1981.
[3] S. McFarling, "Combining branch predictors," *TR TN-36, DEC*, 1993.
[4] E. Sprangle, R. Chappell, M. Alsup, and Y. Patt, "The agree predictor: A mechanism for reducing negative branch history interference," *ISCA*, pp. 284–291, 1997.
[5] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," *MICRO*, pp. 281–290, 1997.
[6] T. Yeh and Y. Patt, "Alternative implementations of two-level adaptive branch prediction," *ISCA*, pp. 124–134, 1992.
[7] G. Loh, "Revisiting the performance impact of branch predictor latencies," *ISPASS*, pp. 59–69, 2006.
[8] E. Hao, P. Chang, and Y. Patt, "The effect of speculatively updating branch history on branch prediction accuracy, revisited," *MICRO*, pp. 228–232, 1994.
[9] D. Jimenez, "Delay-sensitive branch predictors for future technologies," Ph.D. dissertation, 2002, univ. Texas at Austin.
[10] S. Almukhaizim, T. Verdel, and Y. Makris, "Cost-effective graceful degradation in speculative processor subsystems: The branch prediction case," *ICCD*, pp. 194–197, 2003.
[11] S. Almukhaizim, P. Petrov, and A. Orailoglu, "Low-cost, software-based self-test methodologies for performance faults in processor control subsystems," *CICC*, pp. 263–266, 2001.
[12] S. Almukhaizim, P. Petrov, and A. Orailoglu, "Faults in processor control subsystems: Testing correctness and performance faults in the data prefetching unit," *ATS*, pp. 319–324, 2001.
[13] T. Sato, "First step to combining control and data speculation," *IWIA*, pp. 53–60, 1998.
[14] J. Gonzalez and A. Gonzalez, "The potential of data value speculation to boost ILP," *ICS*, pp. 21–28, 1998.
[15] R. Littin, "Data and control speculative execution," *NZCSRSC*, 1999.
[16] N. Karimi, M. Maniatakos, Y. Makris, and A. Jas, "On the correlation between controller faults and instruction-level errors in modern microprocessors," *ITC*, pp. 24.1.1–24.1.10, 2008.
[17] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," *DSN*, pp. 61–70, 2004.
[18] 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.
[19] D. Burger, T. M. Austin, and S. Bennett, "Evaluating future microprocessors: The simplescalar tool set, Tech. Rep. CS-TR-1996-1308, 1996. [Online]. Available: citeseer.ist.psu.edu/burger96evaluating.html
[20] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and Y. Makris, "Instruction-level impact comparison of RT- vs. gate-level faults in a modern microprocessor controller," *VTS*, pp. 9–14, 2009.
[21] J. C. Baraza, J. Gracia, S. Blanc, D. Gil, and P. J. Gil, "Enhancement of fault injection techniques based on the modification of VHDL code," *IEEE TVLSI*, vol. 16, no. 6, pp. 693–706, 2008.