Comparing the Effects of Intermittent and Transient Hardware Faults on Programs

Jiesheng Wei, Layali Rashid, Karthik Pattabiraman and Sathish Gopalakrishnan
Department of Electrical and Computer Engineering
The University of British Columbia, Canada
{jwei, lrashid, karthikp, sathish}@ece.ubc.ca

***Abstract*—The trends of shrinking device geometries, lower voltages and higher frequencies in modern processors are expected to increase the rate of intermittent faults. This requires the design of software that are resilient to intermittent faults. There has been substantial research on software systems that are resilient to transient faults. However, it is unclear whether the impact of intermittent faults on programs is similar to that of transient faults. This is important for deciding if we need novel techniques for tolerating intermittent faults in software. In this study, we attempt to answer this question by comparing the effects of intermittent and transient hardware faults on programs through fault-injection experiments performed in a micro-architectural simulator for a simple five-stage pipelined processor. We also investigate whether the differences (if any) vary with the length (i.e., duration in cycles) of the fault and with the micro-architectural unit in which the fault originates. The result show that intermittent faults' impact on programs are significantly different from those of transient faults, and that the difference depends both on the length of the fault and the fault's origin. Therefore, existing software techniques for ensuring resilience from transient faults may not be sufficient for intermittent faults, and new techniques are needed.**

***Index Terms*—Micro-architectural-level fault injection; Intermittent fault; Transient fault;**

I. Introduction

Intermittent faults are error bursts that occur repeatedly at the same micro-architectural location in a processor. Such faults occur due to process variation, manufacturing residuals, in-progress wear-out, and voltage and temperature fluctuations [1][2]. The trends of shrinking device geometries, lower voltages and higher frequencies in modern processors have exacerbated these factors, and hence the rate of intermittent faults is expected to increase [3]. Further, such faults cannot be entirely masked at the hardware level due to constraints of power and area. Therefore, there is a compelling need to design software systems that are resilient to intermittent faults.

An important first step in designing fault-tolerant software is a study of the effects of faults on programs. Many studies have examined the impact of transient faults on software programs [4][5]. Transient faults are caused by a variety of factors such as ionizing radiation, and unlike intermittent faults, do not usually reoccur at the same location. However, to our knowledge, there has been no study on how intermittent faults affect software systems, and whether their effects on programs are different from those of transient faults.

This paper studies the effects of intermittent faults in the processor on programs executing on it. We characterize intermittent faults based on their lengths and the micro-architectural unit at which they originate. We also study two types of intermittent faults - namely, stuck-at-zero and stuck-at-one faults. The main question we ask in this paper is ''Do intermittent faults affect programs differently from transient faults, and if so, what are the differences?''. This question is important to design programs that are resilient to intermittent faults. For example, if intermittent faults cause a higher percentage of crashes in programs, then it makes sense to focus on the design of efficient checkpointing and recovery techniques. On the other hand, if such faults cause a higher percentage of Silent Data Corruptions (SDCs), then application-specific semantic checks may be required.

Because studying hardware faults in a real system is difficult, fault injection frameworks are widely used for this purpose. Performing fault injection at the software level [6][7] can help us understand fault propagation in application programs. However, such injections may not be an accurate representation of the the faults in the underlying processor. On the other hand, fault injections at the circuit or gate levels [8] can more accurately represent hardware faults in the processor. However, these studies are often unable to evaluate the effects of faults on application behavior as they are challenging to scale to real applications.

This paper introduces a software framework to inject both intermittent and transient faults at the micro-architectural level and compare their impact on application programs. Injecting faults at the micro-architectural level represents a reasonable trade-off between accuracy and scalability and is well-suited to the needs of the study. Further, fault injection at the micro-architectural level enables us to compare fault effects across different micro-architectural units. Through this study, we investigate the three research questions below:

1) Do intermittent faults differ significantly in their impact on software programs?
2) How do the differences vary with the length (i.e., duration in cycles) of the fault?
3) How do the differences vary with the micro-architectural unit in which the fault originates?

This work was supported in part by the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC and by an NSERC Discovery Grant.

The results show that intermittent faults differ significantly from transient faults. We also find that the length of the fault, the fault type and unit of origin have significant effects on the differences. From this, one can conclude that we need to rethink resilient software design for intermittent faults, and that existing mechanisms for resilience to transient faults may be insufficient for intermittent faults. This study also motivates the choice of appropriate software fault models for representing intermittent faults in the processor.

II. Fault Model

This section describes the fault model(s) considered in the study and how the faults are represented. As mentioned before, we consider intermittent hardware faults in processors.

For this study, we consider a simple five-stage pipeline RISC processor adapted from Hennessey and Patterson [9]. Figure 1 shows an architectural block diagram of the processor and its signals. Although this is a simple processor, it is representative of many micro-controllers used in embedded systems. The signals that we consider for fault-injection in the processor are shown in italics in the figure. Table I presents a detailed description of the signals considered for fault-injections. It can be seen from the table that our fault model covers most signals in the execution data-path in the processor. However, we do not consider faults that occur in (1) memory and cache, as we assume that these are ECC-protected, (2) speculative logic signals, as these are unlikely to impact the program [10], (3) floating-point unit, as the fault's classification is dependent on the application's tolerance of approximations in floating point and (4) certain control-logic signals such as *memory acknowledge*, due to limitations of our simulation infrastructure. We also assume that only one signal is affected by the intermittent fault during the execution of a program.

We model intermittent faults as stuck-at-zero or stuck-at-one faults for specific durations of time. Prior studies have observed intermittent stuck-at faults that activate and de-activate repeatedly [11]. However, we assume that there is only one intermittent fault burst during the execution of the program. This is because intermittent faults are typically rare-events, and it is unlikely that they will recur within the same program's execution. Further, we assume that a signal affected by an intermittent fault is stuck-at-zero or stuck-at-one for the entire duration of the fault. In other words, we assume that de-activation time for an intermittent fault is zero during its occurrence. This is consistent with prior work which finds that deactivation time in bus- and memory- faults does not significantly impact the failure rate of programs [11].

Although transient faults may be modeled in a number of ways, we assume that they result in single bit flips and last for one clock cycle. This is consistent with prior work [5][6].

III. Experimental Setup

In this section, we discuss the details of the fault-injection framework for our experiments.

TABLE I
Fault locations and description

Fault location	Description
PC	Program counter of Instruction Fetch (IF) stage
Opcode	Opcode decoded in Instruction Decode (ID) stage
Operand	Operand decoded in ID stage. We assume all of the 6 operand signals shown in Figure 1 are equally likely to be faulty. Therefore, we randomly choose one of them as the fault injection location in each run.
Mux_a	Output of operand multiplexer a, which selects data from register or current PC to output
Mux_b	Output of operand multiplexer b, which selects data from register, immediate number, target address, offset address to output
ALU_o	Output of Arithmetic Logic Unit (ALU)
MULT_o	Output of Multiplication/Division Unit
LSU_adr	Address output of Load Store Unit (LSU)
Ld_data	Load data of Load Store Unit (LSU)
St_data	Store data of Load Store Unit (LSU)

A. Experimental infrastructure

We have implemented the fault-injection framework in the *sim-outorder* processor simulator from the *SimpleScalar* family of simulators using the PISA instruction set. The SimpleScalar micro-architectural simulators model an architecture that is a close derivative of the MIPS architecture [12]. The configuration parameters that we use for running the simulator are shown in Table II. These model the simple processor in Figure 1.

We have manually examined the source code of the simulator and identified the variables in the simulator that correspond to the signals in Table I. For each signal identified in the table, there is a unique variable (or a set of variables) in the *sim-outorder* simulator that represents the signal's value. We wrap the definition of these variables in the simulator with a custom fault-injection function to represent the injection of a fault in the corresponding signal. This function injects the appropriate type of fault in the signal for a duration specified through a configuration file (e.g., stuck-at-zero fault for 800 cycles starting from cycle 10216). Only one bit in the selected signal is injected, and it is randomly chosen in each run. Only one fault is injected in each run, and the starting cycle for the fault is also chosen randomly from the total execution cycles executed by the program under a fault-free execution (this value is obtained through profiling).

TABLE II
Simulator Configuration Parameters

Configuration Parameter	**Value**
Fetch/decode/execute/commit rate	1 per cycle
Branch prediction type	Perfect prediction
Register update unit (RUU) size	16
Load/store queue (LSQ) size	8
Register file	32 integer registers, 32 float point registers
Instruction/Data L1	16KB each
L1 hit latency	1 clock cycle
L2 (Unified)	256KB
L2 hit/miss latency	6/18 clock cycles

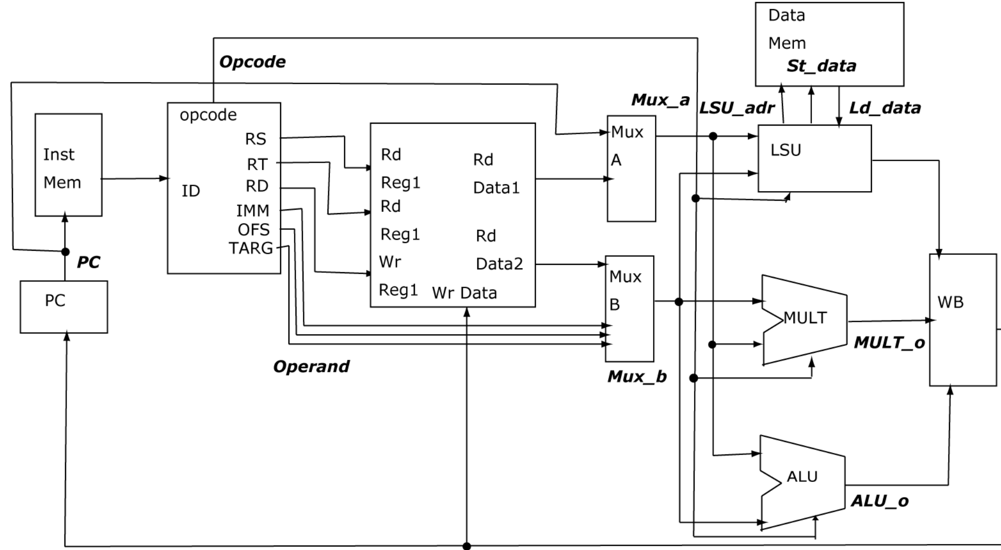

Fig. 1. Block Diagram of MIPS micro-architecture

B. Failure Detection

We classify the outcome of the injection into one of four categories: (1) crash, (2) hang, (3) silent data corruption (SDC) and (4) no effect. Table III presents a detailed classification of the program's behaviour under faults and how our simulator detects the erroneous behavior and classifies it. We modified the SimpleScalar simulator to detect the errors (as done by prior work [13]) and classify them. We need to execute the program once without faults to obtain a golden run.

TABLE III
MEASUREMENT OF DIFFERENT TYPES OF FAILURES

Failure type	Simulation detection mechanism
Crash: Invalid memory access	Check on memory access
Crash: Memory alignment error	Check on memory access
Crash: Division-by-zero	Check before division operation
Crash: Integer overflow	Check after every integer operation
Crash: Invalid instruction	Check instruction validity before instruction execution
Crash: System call error	Check in system call
Hang	Check if program execution time is substantially longer than golden run (5 times as much)
SDC	Check if program's output deviates from the golden run

C. Benchmark Information

We use the Siemens suite of benchmark programs for our evaluation [14]. These programs have been extensively used by the software testing community, and range in size from 100 to 1000 lines of C code. Table IV shows the runtime characteristics of the Siemens programs.

We do not use larger benchmark suites such as the SPEC2006 suite [15], because we need to perform hundreds of thousands of fault-injection experiments in this study. Programs in the SPEC suite execute millions of dynamic instructions, and hence take too long to execute in a detailed micro-architectural simulator for that many injections. The fault-injection experiments in our study completed within one day on a quad-core processor.

TABLE IV
CHARACTERISTICS OF PROGRAMS IN SIEMENS BENCHMARK SUITE

Benchmark	Number of instructions committed	Number of load instructions	Number of store instructions	Number of branch instructions
print_tokens	27,273	4,630	6,985	5,047
print_tokens2	25,093	3,542	6,490	4,883
replace	12,590	1,666	3,981	2,257
schedule	162,497	36,570	22,586	35,108
schedule2	239,993	47,173	35,138	51,284
tcas	8,778	759	3,499	1,479
tot_info	26,543	4,418	6,318	5,179

D. Experimental Parameters

Table V summarizes the ranges of parameters used in our experiments. We conduct 1000 fault-injection runs for each combination of fault type, fault location and fault length. Each run injects one fault. Thus,we inject a total of **1,071,000 faults** ((2 fault types$\times$9 fault signals$\times$8 fault lengths$\times$1000 for intermittent faults $+1$ fault type$\times$9 fault signals$\times$1 fault length$\times$1000 for transient faults)$\times$7 benchmarks).

Note however that not all injected faults may be activated (i.e., read in the system). We measure the activation rate of the faults by instrumenting the fault-injected locations in the simulator, and use the number of activated faults as the denominator when calculating percentages in this study.

IV. RESULTS

This section describes the results of our experiments and is organized according to the research questions in Section I.

TABLE V
PARAMETERS FOR A FAULT INJECTION RUN

Experimental Parameter	Values
Fault type	stuck-at-zero, stuck-at-one for intermittent faults, and bit-flips for transient faults
Fault location	Signals in Table I except *MULT_o*, since there are no multiplication operations in many programs
Fault bit	Randomly chosen from bits of the selected signal
Fault-injection start time	Randomly chosen from the total clock cycles executed by the program
Fault length	2, 5, 10, 25, 50, 100, 300 or 800 clock cycles for intermittent faults

A. Impact of Intermittent Faults on Programs

Table VI presents the average failure rate across different benchmarks for transient faults, intermittent stuck-at-zero faults, and intermittent-stuck-at-one faults. The length of the intermittent faults for this experiment is 50 clock cycles.

TABLE VI
AVERAGE FAILURE PERCENTAGE ACROSS DIFFERENT PROGRAMS FOR INTERMITTENT FAULTS AND TRANSIENT FAULTS

Benchmarks	Transient faults			Intermittent stuck-at-zero faults			Intermittent stuck-at-one faults		
	crash	hang	SDC	crash	hang	SDC	crash	hang	SDC
print_tokens	47	0	5	30	1	5	71	1	7
print_tokens2	46	0	3	26	1	4	70	1	5
replace	44	1	3	30	0	2	65	1	5
schedule	39	1	8	45	1	6	69	1	9
schedule2	47	0	7	46	1	5	68	1	6
tcas	42	0	1	28	0	1	70	1	1
tot_info	49	0	4	34	6	4	70	2	7

The results in the table are as follows:

- Crashes are the dominant outcome for all three fault categories (30 to 71%), followed by SDCs (1 to 9%), followed by hangs (0 to 6%).
- Intermittent stuck-at-one faults incur higher percentages of crashes when compared with intermittent stuck-at-zero faults. This is because a program usually grows from small addresses to big addresses and hence, a larger address is more likely to be out of the memory bounds of the program. Because the main reason of crashes for all the faults except those originating from *Opcode* is invalid memory access, and stuck-at-one faults change original address to a larger value while stuck-at-zero faults change it to a smaller value, the former will incur higher percentages of crashes.
- Intermittent stuck-at-zero faults incur a lower percentage of crashes compared to transient faults for all but two programs, *schedule* and *schedule2*. This is because in general transient faults are guaranteed to change the value of the signal (as they flip a bit), while an intermittent stuck-at-zero fault will only change the signal if the chosen bit is 1. As mentioned before, most bits in the injected signals are 0s and hence unlikely to change due to an intermittent stuck-at-zero fault. However, both *schedule* and *schedule2* execute significantly higher numbers of instructions than the other programs (see Table IV), and hence, there is a higher likelihood of the intermittent fault persisting and propagating in these programs, thus leading to crashes. This shows that the number of instructions executed by a benchmark can also affect fault propagation characteristics.
- SDCs do not differ by much between intermittent stuck-at-zero faults and transient faults. However, the occurrence of SDCs for intermittent stuck-at-one faults is slightly higher than the corresponding values for either transient faults or intermittent stuck-at-zero faults. Hangs do not change much across the fault types.

B. Effect of Intermittent Fault Lengths

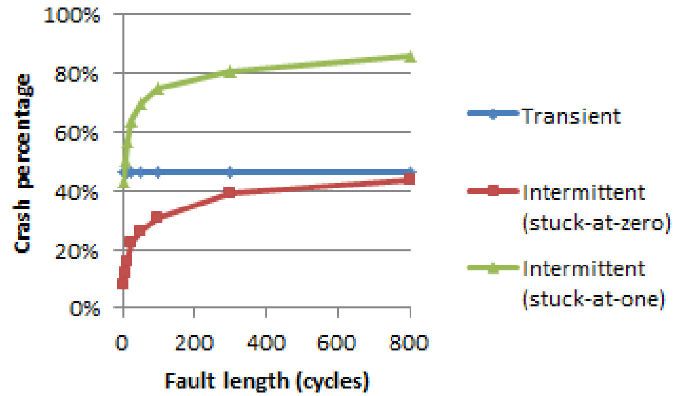

Fig. 2. Average program crash percentage of transient faults and intermittent faults with the increase of intermittent fault length (Note that the transient faults last exactly 1 clock cycle, i.e., have a length of 1.)

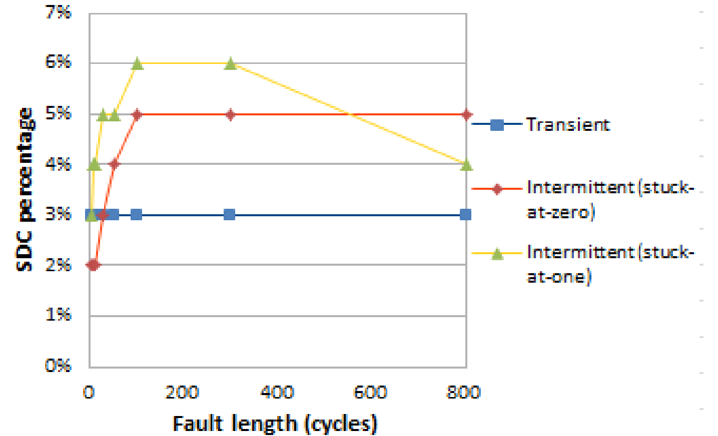

Fig. 3. Average program SDC percentage of transient faults and intermittent faults with the increase of intermittent fault length (Note that the transient faults last exactly 1 clock cycle, i.e., have a length of 1)

In this section, we focus on one benchmark program, *print_tokens2*, to understand the variation in the effects of intermittent faults with the length of the fault. Due to space

constraints, we are unable to present the results for the other benchmarks, but we have observed similar trends across all benchmarks.

Figures 2 and 3 show the variation in the percentage of crashes and SDCs respectively. We do not present results for hangs because the percentages of hangs is very small (less than 2%) and there is no clear trend because of the relatively small number of samples.

The main results are as follows:

- The percentage of crashes increases as the fault length of intermittent faults increases. However, the growth flattens out as the fault length increases, as the crashes caused by faults of longer fault length overlap significantly with those covered by the faults of shorter fault length. Moreover, intermittent stuck-at-one faults have higher crash percentages than that of stuck-at-zero faults, though their difference stays constant as fault length increases. This is consistent with the results in Section IV-A.
- Transient faults' crash percentage does not change as they last only one cycle. The intermittent stuck-at-one faults' crash percentages equal the transient faults' crash percentages when the fault length is 3 clock cycles, while for intermittent stuck-at-zero faults, they become equal when the fault length is over 800 clock cycles.
- For both types of intermittent faults, the SDC percentage first increases, and then saturates, before decreasing (for intermittent stuck-at-one faults) . This is because, as the fault length increases, some of faults that may have been benign lead to SDCs, thus increasing the percentage of SDCs. However, as the fault length increases even further, some faults that would have caused SDCs cause the the program to crash, thus decreasing the SDC percentage.
- Comparing the SDCs for intermittent stuck-at-zero faults with those of intermittent stuck-at-one faults, we find that the latter is initially higher than the former, but as fault length increases beyond 300 clock cycles, the percentage of SDCs for the latter drops below the former. The first effect is because stuck-at-one faults are more likely to impact the program than stuck-at-zero faults, and hence a fault that would be benign in the stuck-at-zero case causes an SDC in the stuck-at-one case. However, if the fault length increases beyond a point (i.e., 300 cycles), the same fault is more likely to cause a crash in the stuck-at-one case, while it is more likely to cause an SDC in the stuck-at-zero case.

C. Effect of Intermittent Fault Origin

To understand the effects of faults originating in different micro-architectural units, we classify the outcomes of the fault-injection experiments based on the signal that is injected. Due to space constraints, we present results only for the *print_tokens2* program as in the previous section. In this experiment, we fix the type of intermittent fault to be stuck-at-zero and fault length to be 50 cycles as we did before. Figure 4 shows the effects of intermittent faults and transient faults across different signals.

The main take-away from the data graphs is that for all signals except *Opcode* and *Mux_b*, the percentage of crashes caused by transient faults is higher than those caused by intermittent faults. This is because the majority of bits in these signals are *always* zeroes, and hence an intermittent stuck-at-zero fault is unlikely to change their value. On the other hand, a transient fault is guaranteed to change their values by flipping a bit (however, not all such changes impact the program, which is why the percentages of failures is not 100%). For the *Opcode* signal though, every bit *can* be 1, and hence a stuck-at-zero fault is more likely to change the signals. Therefore, they have a more pronounced effect than transient faults, which last for only one cycle. We need further investigation to understand why this is so for the *Mux_b* signal.

Figure 4 also shows that the difference of the crash percentages between transient and intermittent faults varies across different units, which means that the vulnerability of a unit to transient faults is often different from its vulnerability to intermittent faults.

V. Related Work

Fault injection framework at micro-architectural level: Fault injection frameworks built on top of micro-processor simulators are widely used to study different types of hardware faults. Li et al. [16] inject permanent faults in three micro-architectural units (ALU, Address Generation Unit and decoder) using a software simulator to analyze the accuracy of micro-architectural fault model versus a gate-level fault model. Karimi et al. [17] inject faults into different micro-architectural-level control logic signals to study their impact on instruction execution. Both studies have similar goals as our study. However, they cover only a small subset of micro-architectural signals. The transient fault injection framework proposed in [18] performs a comprehensive fault-injection study in a Verilog model of the processor and studies the impact of faults on processor state and programs. However, their work breaks a program into many breakpoints and treats an injected fault at a breakpoint as benign if the fault does not propagate to the next breakpoint. In contrast, we consider the effects of faults on the entire program, which is more representative of their final outcome. Further, none of the three studies consider intermittent faults, which is our focus.

Studies of comparison between different fault types: Gracia et al. [11] compare the impact of intermittent faults in registers, buses and memory on application programs with those of transient faults and permanent faults. In later work [19], they extend the comparison to determine which fault type is more likely to be detected in a fault tolerant system and handled by the existing recovery mechanisms. However, neither study consider impact of faults in the signals of the processor's pipeline stages, which our study does.

Other studies of intermittent faults: Rashid et al. [7] study the impact of intermittent faults at the program level by gathering the fault-free program execution and predicting the propagation of errors using the program's trace. However, because they inject faults at the program level, they are unable

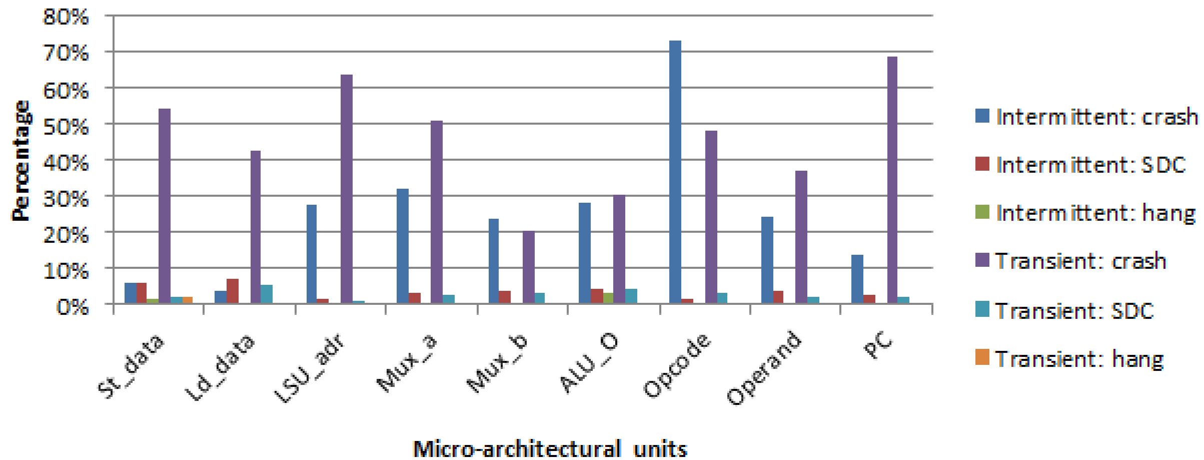

Fig. 4. Effect of stuck-at-zero intermittent faults and transient faults across different units in the processor. Fault length for intermittent faults is 50 cycles.

to accurately model intermittent faults in individual micro-architectural units. Pan et al. [20] propose a new metric Intermittent Vulnerability Factor (IVF) to characterize the vulnerability of different micro-architectural units to intermittent faults. Similar to us, their goal is to study the differences in the sensitivites of different micro-architectural units to intermittent faults. However, they use analytical modeling methods which are known to be less accurate than a fault-injection based approach, which we use in our study.

VI. Conclusions and Future Work

This paper builds a micro-architectural fault injection framework to compare the impact of transient faults and intermittent faults on application programs. Results show that transient faults and intermittent faults have substantial differences in the percentage of crashes they cause in programs. However, the differences are less marked for SDCs and hangs. We also find that the differences are dependent on (1) the length of the intermittent fault, (2) the fault type, and (3) its origin in terms of the micro-architectural unit.

Future work will attempt to fully understand the differences among transient and intermittent faults better, and to build a model for representing intermittent faults at the program level. We will also investigate techniques to build software systems that are resilient to intermittent faults based on the results of this study. Finally, we will extend the study to more complex processors, including those supporting out-of-order execution.

References

[1] S. Borkar, "Microarchitecture and design challenges for gigascale integration," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, 2004.

[2] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Adapting to intermittent faults in multicore systems," *SIGARCH Computer Architecture News*, vol. 36, 2008.

[3] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, vol. 23, 2003.

[4] E. Czeck and D. Siewiorek, "Effects of transient gate-level faults on program behavior," in *Proceedings of 20th International Symposium on Fault-Tolerant Computing*, 1990.

[5] J. Ohlsson, M. Rimen, and U. Gunneflo, "A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog," in *Proceedings of Intl. Symposium on Fault-Tolerant Computing*, 1992.

[6] S. Han, K. Shin, and H. Rosenberg, "DOCTOR: an integrated software fault injection environment for distributed real-time systems," in *Proceedings of International Computer Performance and Dependability Symposium*, 1995.

[7] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, "Modeling the propagation of intermittent hardware faults in programs," 2010.

[8] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: a methodology and some applications," *IEEE Transactions on Software Engineering*, vol. 16, 1990.

[9] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/software Interface*, 4th ed. Morgan Kaufmann, 2008.

[10] C. Weaver and T. Austin, "A fault tolerant approach to microprocessor design," in *Proceedings of International Conference on Dependable Systems and Networks*, 2001.

[11] J. Gracia, L. Saiz, J. Baraza, D. Gil, and P. Gil, "Analysis of the influence of intermittent faults in a microcontroller," in *Proceedings of Workshop on Design and Diagnostics of Electronic Circuits and Systems*, 2008.

[12] D. Burger, T. M. Austin, and S. Bennett, "Evaluating future microprocessors: the SimpleScalar tool set," University of Wisconsin-Madison, Computer Science Department, Tech. Rep., 1996.

[13] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "Application-based metrics for strategic placement of detectors," in *Proceedings of Pacific Rim International Symposium on Dependable Computing*, 2005.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of Intl. Conference on Software Engineering*, 1994.

[15] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, 2006.

[16] M. Li, P. Ramachandran, U. Karpuzcu, S. Hari, and S. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *Proceedings of IEEE 15th International Symposium on High Performance Computer Architecture*, 2009.

[17] N. Karimi, M. Maniatakos, A. Jas, and Y. Makris, "On the correlation between controller faults and instruction-level errors in modern microprocessors," in *Proceedings of International Test Conference*, 2008.

[18] S. Kim and A. Somani, "Soft error sensitivity characterization for microprocessor dependability enhancement strategy," in *Proceedings of International Conference on Dependable Systems and Networks*, 2002.

[19] J. Gracia-Moran, D. Gil-Tomas, L. Saiz-Adalid, J. Baraza, and P. Gil-Vicente, "Experimental validation of a fault tolerant microcomputer system against intermittent faults," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks*, 2010.

[20] S. Pan, Y. Hu, and X. Li, "IVF: Characterizing the vulnerability of microprocessor structures to intermittent faults," in *Proceedings of IEEE /ACM Conference on Design, Automation and Test in Europe*, 2010.