On the Impact of Hardware Faults - An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions

Domenico Di Leo*, Fatemeh Ayatolahi**, Behrooz Sangchoolie**, Johan Karlsson**, Roger Johansson**

*Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Naples, Italy
**Department of Computer Science & Engineering, Chalmers University of Technology, SE-41296, Gothenburg, Sweden
domenico.dileo@unina.it,
{fatemeh.ayatolahi,behrooz.sangchoolie,johan,roger}@chalmers.se

Abstract. Technology scaling of integrated circuits is making transistors increasingly sensitive to process variations, wear-out effects and ionizing particles. This may lead to an increasing rate of transient and intermittent errors in future microprocessors. In order to assess the risk such errors pose to safety critical systems, it is essential to investigate how temporary errors in the instruction set architecture (ISA) registers and main memory locations influence the behaviour of executing programs. To this end, we investigate – by means of extensive fault injection experiments – how such errors affect the execution of four target programs. The paper makes three contributions. First, we investigate how the failure modes of the target programs vary for different input sets. Second, we evaluate the error coverage of a software-implemented hardware fault tolerant technique that relies on triple-time redundant execution, majority voting and forward recovery. Third, we propose an approach based on assembly language metrics which can be used to correlate the dynamic fault-free behaviour of a program with its failure mode distribution obtained by fault injection.

Keywords: microprocessor faults, fault injection, dependability assessment, software-implemented hardware fault tolerance, failure mode distributions.

1 Introduction

Technology and voltage scaling is making transistors increasingly susceptible to process variations, wear-out effects, and ionizing particles [1]. This is expected to increase the rate of transient, intermittent and permanent transistors faults in future integrated circuits. Modern microprocessors are therefore being equipped with increasingly sophisticated hardware mechanisms for fault tolerance, error masking and error detection. However, since such mechanisms cannot provide perfect error coverage, and due to the fact that the number of transistors per chip is steadily increasing, it is

likely that future microprocessors will exhibit an increasing rate of incorrect program executions caused by hardware related errors.

A cost-effective way of reducing the risk that such incorrect program executions cause unacceptable or catastrophic system failures is to introduce a layer of software-implemented error handling mechanisms. Numerous software techniques for detecting and masking hardware errors have previously been proposed in the literature [2, 3]. The effectiveness of these techniques are often evaluated, or benchmarked, by means of fault injection experiments that measure their ability to detect or mask single or multiple bit errors (bit flips) in CPU registers and main memory [4]. Bit flipping is used to emulate the effect of single event upset (SEU) errors caused by ionizing particles. The error coverage for software-implemented error handling techniques often depends on the input processed by the target system. Thus, to assess the variability in error coverage, it is essential to conduct fault injection experiments with different inputs [5, 6].

This paper presents the results of extensive fault injection experiments with four programs where single bit errors were injected in CPU registers and main memory of the target systems. The aim of the study is to investigate how error coverage varies for different inputs. We conducted experiments with programs protected by triple-time redundant execution with forward recovery [7], and programs without software-implemented hardware fault tolerance (SIHFT). In addition, we propose a technique for identifying input sets that are likely to cause the measured error coverage to vary.

The remainder of the paper is organized as follows. We describe the target workloads in Section 2 and the TTR-FR mechanism in Section 3. The fault injection experimental setup is described in Section 4. The analysis of the extensive fault injections conducted on the workloads with/without TTR-FR mechanism is presented in Section 5. Based on the obtained results, we present the input selection approach in Section 6.

2 Target Workloads

In this section, we present the four workloads used in our set of experiments: secure hash algorithm (SHA), cyclic redundancy check (CRC), quick sort (Qsort), and binary string to integer convertor (BinInt). SHA is a cryptographic hash function which generates a 160-bit message digest. We use SHA-1 algorithm which is adopted in many security protocols and applications such as SSL, SSH and IPsec. The CRC that we use is a software implementation of CRC 32-bit polynomial which is mostly used to calculate the end-to-end checksum. Qsort is a recursive implementation of the well-known quick sort algorithm, which is also used as a target program for fault injection experiments in [6, 8]. Finally, BinInt converts an ASCII binary string, 1s and 0s, into its equivalent integer value.

Even though the implementation of our workloads can be found in the MiBench suite [9], we only take CRC and BinInt from this suite. For the quick sort algorithm, the MiBench implementation uses a built-in C function named qsort whose source code is not available. This prevents us from performing detailed analysis. Furthermore, the MiBench implementation of SHA uses dynamic memory allocation which

is not necessary for an embedded system. Thus, we adopt another implementation of SHA[1]. The structure of these synthetic workloads profoundly differs in terms of lines of source code (LOC), number of functions, input types and executed assembly instructions. BinInt is the smallest workload with 7 LOC and is made of one function with one loop, whereas SHA measures 125 LOC and has 5 functions.

2.1 Input Sets

Nine different inputs are selected for each workload. The combination of an input and a workload is called an *execution flow*. Thus, for each workload, we have conducted experiments for 9 execution flows. On the basis of the length of the inputs, we group SHA and CRC execution flows into three categories of small, medium, and large inputs, see Table 1. These categories are chosen to represent input lengths that are common in real applications. For Qsort, the input vector consists of 6 integers. The execution flows use the same 6 integers with different permutations. In this way, the inputs cover a range of possibilities, including sorted, mostly sorted, partly sorted, and unsorted, see Table 2. The input of BinInt is a random string of 1s and 0s. Since an integer is a 32-bit data type, the length of the input string is limited to 32 characters.

Table 1. The input space for CRC *(left table)* and SHA *(right table)* execution flows

Category	Input length (characters)	Execution flow
Small	0	CRC-1
	1	CRC-2
	2	CRC-3
Medium	10	CRC-4 & CRC-5
	46	CRC-6 & CRC-7
Large	99	CRC-8 & CRC-9

Category	Input length (characters)	Execution flow
Small	0	SHA-1
	1	SHA-2
	2	SHA-3
Medium	10	SHA-4 & SHA-5
	60	SHA-6 & SHA-7
Large	99	SHA-8 & SHA-9

Table 2. The input space for Qsort *(left table)* and BinInt *(right table)* execution flows

Category	# of sorted elements	Execution flow
Sorted	6	Qsort-1
Mostly sorted	4	Qsort-2 & Qsort-3
Partly sorted	3	Qsort-4 & Qsort-5
	2	Qsort-6 & Qsort-7
Unsorted	0	Qsort-8 & Qsort-9

Category	Input length (characters)	Execution flow
Small	0	BinInt-1
	9	BinInt-2 & BinInt-3
Medium	16	BinInt-4 & BinInt-5
	24	BinInt-6 & BinInt-7
Large	31	BinInt-8 & BinInt-9

[1] http://www.dil.univ-mrs.fr/~morin/DIL/tp-crypto/sha1-c

3 Software-Implemented Hardware Fault Tolerance (SIHFT)

In addition to the basic version of the workloads, we conducted experiments on the triple time redundant execution with forward recovery (TTR-FR) [7]. In TTR-FR, the target workload is executed three times and the result of each run is compared with the other two runs using a software-implemented voter. If only one run of the program generates a different output, the output of the other two runs will be selected (majority voting). In case the workload is state-full, the state of the faulty run moves forward to a fault-free point (forward recovery). If none of the outputs match, then error detection is signaled.

The non-fault tolerance version of the workloads consists of three major code blocks; startup, main function, and core function. In the TTR-FR implementation we add the voter to the main function to perform the majority voting. The core function, which is called three times from the main function, performs the foremost functionality of each workload. As an example, in Qsort, the sorting procedure is done in the Qsort's core function, whereas in CRC, the core function is responsible for the checksum calculations.

4 Experimental Setup and Fault Model

The workloads are executed on a Freescale MPC565 microcontroller, which uses the PowerPC architecture. Faults are injected into the microcontroller via a Nexus debug interface using *Goofi-2* [10], a tool developed in our research group. This environment allows us to inject faults, bit flips, into instruction set architecture (ISA) registers and main memory of the microcontroller. Ideally, the fault model to adopt for this evaluation should exhibit real faults, i.e., it should account for multiple and single bit flips. However, there is no commonly agreed model for multiple bit flips. Thus, we adopt the single bit flip model as it has been done in other studies [11, 2, 3, 10].

The faults are injected in the main memory (stack, data, etc.) and all CPU registers used by the execution flows. The registers include general purpose registers, program counter register, link register, integer exception register, and condition register. As the machine code of our workloads is stored in a Flash memory, it cannot be subjected to fault injection. We define fault in terms of time-location pair, where the location is a randomly selected bit in the memory word or CPU register, while the time corresponds to the execution of a given machine instruction (i.e., a point in the execution flow). Indeed, we make use of a pre-injection analysis [8] which is included in Goofi-2. In this way, the fault injection takes place on a register or memory location, just before it is read by the executing instruction. A fault injection *experiment* consists of injecting one fault and observing its impact on a workload. A fault injection *campaign* is a series of fault injection experiments with a given execution flow.

5 Experimental Results

In this section, we present the outcomes of fault injection campaigns conducted on the 4 workloads. We carried out 9 campaigns per workload which resulted in a total of 36 campaigns for the basic version and 36 campaigns for the TTR-FR version. The campaigns consist of 25000 experiments except for CRC campaigns that are subjected to 12000 experiments. The error classification scheme of each experiment is:

- No Impact (NI), errors that do not affect the output of the execution flow.
- Detected by Hardware (DHW), errors that are detected by the hardware exceptions.
- Time Out (TO), errors that cause violation of the timeout[2].
- Value Failure (VF), erroneous output with no indication of failure (silent data corruption).
- Detected by Software (DSW), errors that are detected by the software detection mechanisms.
- Corrected by Software (CSW), errors that are corrected by the software correction mechanisms.

When presenting the results, we also refer to the coverage (COV) as the probability that a fault does not cause value failures, which is calculated in equation (1):

$$COV = 1 - \#VF/N \quad (1)$$

Here N is the total number of experiments, and $\#VF$ is the total number of experiments that resulted in value failure. In addition to the experiments classified as detected by hardware, the coverage includes no impact and timeout experiments. No impact experiments can be the result of internal robustness of the workload; therefore they contribute to the overall coverage of the system. Experiments that are resulted in timeout are detected by Goofi-2. In a real application, watchdog timers are used to detect these types of errors.

5.1 Results for Workloads without Software-Implemented Hardware Fault Tolerance

Table 3 presents failure distributions for all the workloads. Each row shows the percentage of experiments that fall in different error classifications. Due to the large number of experiments (25000 for SHA, BinInt, Qsort and 12000 for CRC), the 95% confidence interval for the measures in this section varies from ±0.08% to ±0.89%.

For SHA and CRC, the percentage of experiments classified as value failures grows as the length of the inputs is increased. If we consider that the value failure is distributed as a normal variable with a mean value equals to the quotient between the number of value failure experiments and the total number of experiments, we can conduct one way *analysis of variance (ANOVA)*. ANOVA is performed by testing the hypothesis H0 which states "*there is no linear correlation between the length of the*

[2] Timeout value is approximately 10 times larger than the execution time of the workload.

input and the percentage of value failure". The results of ANOVA in Table 4 allow us to reject H0 with a confidence of 95%. The reason behind this correlation is that when the length of the input increases, the number of reads from registers and memory locations are increased as well. Therefore, there are more possibilities to inject faults that result in value failure. Obviously, as the value failure increases linearly with the length, the coverage is linearly decreased (Table 3).

Table 3. Failure distribution of all the execution flows (values are in percentage)

Execution flow	NI	VF	DHW	TO	COV
CRC-1	42.7	6.1	48.2	3.0	**93.9**
CRC-2	32.9	17.9	46.7	2.4	**82.1**
CRC-3	28.3	24.3	45.8	1.6	**75.7**
CRC-4	20.8	34.3	44.0	0.8	**65.7**
CRC-5	20.3	35.5	43.6	0.6	**64.5**
CRC-6	17.1	39.6	43.0	0.3	**60.4**
CRC-7	16.6	39.8	43.4	0.2	**60.2**
CRC-8	15.7	41.2	42.7	0.4	**58.8**
CRC-9	16.0	41.9	41.8	0.3	**58.1**
SHA-1	18.9	38.8	41.0	1.4	**61.2**
SHA-2	17.8	40.1	41.0	1.1	**59.9**
SHA-3	17.6	40.8	40.6	1.0	**59.2**
SHA-4	16.8	42.1	39.7	1.4	**57.9**
SHA-5	15.9	43.1	39.4	1.6	**56.9**
SHA-6	11.5	47.1	39.5	1.9	**52.9**
SHA-7	11.4	47.7	39.3	1.6	**52.3**
SHA-8	10.7	48.8	38.8	1.7	**51.2**
SHA-9	10.7	49.1	38.4	1.8	**50.9**
Qsort-1	37.1	12.7	46.8	3.5	**87.3**
Qsort-2	32.8	17.1	46.9	3.2	**82.9**
Qsort-3	31.3	17.7	47.7	3.3	**82.3**
Qsort-4	31.7	18.1	46.8	3.9	**81.9**
Qsort-5	26.5	23.0	47.2	3.3	**77.0**
Qsort-6	29.0	20.7	46.0	4.3	**79.3**
Qsort-7	29.3	20.9	46.3	3.5	**79.1**
Qsort-8	27.2	22.1	46.6	4.2	**77.9**
Qsort-9	25.4	24.2	46.5	4.0	**75.8**
BinInt-1	44.1	3.5	49.9	2.5	**96.5**
BinInt-2	34.9	20.6	41.5	3.0	**79.4**
BinInt-3	34.7	20.6	41.6	3.1	**79.4**
BinInt-4	34.5	20.5	42.0	2.9	**79.5**
BinInt-5	35.3	21.2	40.5	3.0	**78.8**
BinInt-6	35.1	21.0	40.8	3.1	**79.0**
BitInt-7	34.8	21.5	40.5	3.2	**78.5**
BitInt-8	36.7	20.4	40.0	3.0	**79.6**
BinInt-9	35.5	20.9	40.5	3.1	**79.1**

Table 4. Null Hypothesis test results for the workloads

Null Hypothesis(H0)	Input Characteristic	Workload	p-value (α=0.05)	Result	Linear Regression Equation
No linear correlation between VF and input characteristic	Length in characters	CRC	0.029	Reject	VF = 23.20 + 0.22length
		SHA	<0.001	Reject	VF = 40.64 + 0.09length
		BinInt	0.069	Accept	--
	Sorted elements	Qsort	0.053	Accept	--
No linear correlation between DHW and input characteristic	Length in characters	CRC	0.01	Reject	DHW = 45.84 - 0.04length
		SHA	0.034	Reject	DHW = 40.46-0.019length
		BinInt	0.02	Reject	DHW = 45.75 - 0.02length
	Sorted elements	Qsort	0.12	Accept	--
No linear correlation between TO and input characteristic	Length in characters	CRC	0.046	Reject	TO = 1.67 - 0.017length
		SHA	0.37	Accept	--
		BinInt	0.1	Accept	--
	Sorted elements	Qsort	0.18	Accept	--

Qsort and BinInt exhibit a non-linear variation of the value failure with, respectively, the number of sorted elements and the input length (Table 4). For Qsort, this can be explained by considering that in addition to the number of sorted elements, the position of these elements impacts Qsort's behaviour. This causes different number of element comparisons and recursive calls to the core function. This effect is particularly evident for Qsort-4 and Qsort-5. Even though both have 50% of the input elements sorted, there is a difference of 4.85 percentage points between their value failures. Although there is no linear correlation for Qsort, it is notable that the average value failures of the first five execution flows, which have more sorted elements, is 4.22 percentage points lower than the next four execution flows. BinInt, however, is a small program with an input space between 0 to 32 characters; these inputs for such a small application do not cause a significant variation in the failure distribution.

Results in Table 4 show that the proportion of failures detected by the hardware exceptions is almost constant for a given workload (the coefficient is 0.019 for SHA, 0.04 for CRC, and 0.02 for BinInt). Analogously, the proportion of experiments classified as timeout is almost constant for all the workloads.

It is worth noting that the startup code may vary in different systems. We therefore show the trend of value failures with/without the startup block in Fig. 1. We can see that the trends in the two diagrams are similar which is due to the fact that the startup code consists of significantly fewer lines of code compared to the other blocks.

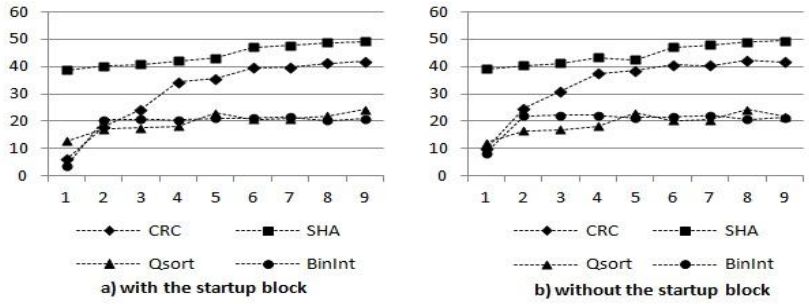

Fig. 1. The percentage of value failures for different execution flows of each workload

5.2 Results for Workloads Equipped with TTR-FR

Table 5a presents the average results for the 9 execution flows of each workload. The percentage of value failures for SHA, CRC and BinInt is less than 2%, while for Qsort there is a higher percentage of value failures, about 5%.

The proportion of value failure varies for different code blocks. With respect to the core function, the main contributor to the lack of coverage is faults in the program counter register. These faults change the control flow in such a way that the voter is incorrectly executed or not executed at all. For instance, for the core function of SHA, around 96% of the value failures were caused by faults in the program counter register. Faults injected into the other code blocks, including the voter, are more likely to generate value failures since they are not protected by the TTR-FR. For Qsort, the

relative size of the core function is smaller compared to the other programs. This resulted in only around 57% of the injections in this function, while in the other workloads more than 96% of faults were injected in the core function. This can explain the higher percentage of value failures in Qsort compared to the other workloads.

In order to evaluate the robustness of the voter, we conducted exhaustive fault injections (i.e., we inject all possible faults) in the voter of each workload, see Table 5b. It is notable that even though TTR-FR mechanism decreases the percentage of value failure, the voter is one of the main contributors to the occurrence of value failure.

The average percentage of errors detected by the hardware exceptions does not vary significantly between the versions extended with TTR-FR and those without this mechanism for SHA, CRC, and BinInt, while it differs about 5% for Qsort.

Table 5. Average failure distributions for workloads with TTR-FR (values are in percentage).

a) All code blocks

Workload	NI	VF	CSW	DSW	DHW	TO	COV
CRC	20.78	1.65	33.43	0.19	43.22	0.73	**98.35**
SHA	14.92	0.76	43.36	0.15	39.00	1.78	**99.24**
Qsort	28.74	5.42	20.37	0.77	41.89	2.79	**94.58**
BinInt	34.69	1.45	20.21	0.09	40.60	2.96	**98.55**

b) Voter code block

Workload	VF
CRC	12.32
SHA	16.60
Qsort	17.05
BinInt	12.32

6 Input Selection

As we demonstrate in this paper, the likelihood for a program to exhibit a value failure due to bit flips in CPU registers or memory words depends on the input to the program. Thus, when we assess the error sensitivity of an executable program by fault injection, it is desirable to perform experiments with several inputs.

In this section, we describe a method for selecting inputs such that they are likely to result in widely different outcome distributions. The selection process consists of three steps. First, the fault-free execution flows for a large set of inputs are profiled using assembly code metrics. We then use cluster analysis to form clusters of similar execution flows. Finally, we select one representative execution flow from each cluster and subject the workload to fault injection. We validate the method by showing that inputs in the same clusters indeed generate similar outcome distributions, while inputs in different clusters are likely to generate different outcome distributions.

6.1 Profiling

We adopt a set of 47 assembly metrics corresponding to different access types (read, write) to registers and memory sections along with various categories of assembly instructions. Specifically, we group the PowerPC instruction set into 6 categories as shown in Table 6. For each group, we define the percentage of execution as the number of times that the instructions of that category are executed out of the total number of executed instructions. These 6 metrics are a proper representative of the metric set

for our workloads. Therefore, these metrics are used as a *signature* for the fault-free run of each execution flow to be used in the clustering algorithm.

Table 6. Assembly metrics corresponding to different instruction categories

Categories	Instructions	Metrics
LOAD (LD)	lbz, li, lwi, lmw, lswi,…	PLD (percentage of load instructions)
STORE (ST)	stb, stub, sth, sthx, stw,…	PST (percentage of store instructions)
ARITHMETIC(AI)	add, subf, divw, mulhw,…	PAI (percentage of arithmetic instructions)
BRANCH (BR)	b, bl, bc, bclr,…	PBR (percentage of branch instructions)
LOGICAL (LG)	and, or, cmp, rlwimi,…	PLG (percentage of logical instructions)
PROCESSOR(PR)	mcrf, mftb, sc, rfi,…	PPR (percentage of processor instructions)

6.2 Clustering

Cluster analysis divides the input set (the execution flow, in our case) into homogenous groups based on the signature of execution flows. We adopted the *hierarchical* clustering [12] due to the fact that unlike other clustering techniques (e.g., K-means), it does not require a preliminary knowledge of the number of clusters. Thus, we can validate a posteriori if the execution flows are clustered as expected. The hierarchical clustering adopted in this work evaluates the distance between two clusters according to the *centroid* method [12]. A similar approach is used in [13].

6.3 Input Selection Results

The clustering technique is applied to normalized values (mean equal to 0 and a variance equal to 1) of the assembly metrics. In the case of non-normalized data, higher weights will be given to variables with higher variances. To prevent this effect, due to the significant variations in the metric values, e.g., the variance of "percentage of load instructions" is orders of magnitude larger than the variance of "percentage of processor instructions", we use the normalized values.

Fig. 2 depicts dendrogram representations of the results of the clustering technique for the non-TTR-FR implementation of SHA, CRC, and Qsort workloads (BinInt has already shown a roughly constant variation in its failure distribution, thus, we exclude it from the clustering analysis). Each dendrogram is read from left to right.

At the first stage of the algorithm, the execution flows of each workload are either grouped in 2-dimension clusters (e.g., SHA-4 and SHA-5) or left isolated (e.g., SHA-1). These groups can be easily linked to characteristics of the inputs in the case of SHA and CRC. Indeed, inputs with the same length (e.g., CRC-9 and CRC-8) or approximately the same length (e.g., CRC-2, CRC-3) belong to the same cluster. However in Qsort, this observation is not verified, since vectors with the same number of sorted elements are placed in different clusters (e.g., Qsort-8 and Qsort-9). At the next stage, different clusters are joined using vertical lines. The positions of these lines indicate the distance at which clusters are joined. In the case of our workloads, the algorithm groups the former clusters together by merging the inputs with "smaller

size" (e.g., SHA-1, SHA-2, SHA-3 with SHA-4, SHA-5) and inputs with "larger size" (e.g., CRC-6, CRC-7 with CRC-8, CRC-9).

In order to validate the results of our approach, we need to show that execution flows with a "similar" failure distribution belong to the same cluster. The same clustering algorithm can be used for identifying the execution flows that are similar in terms of failure distribution. This time, the error categories (VF, NI, DHW, TO) are used instead of the assembly metrics, see Fig. 3. Comparing Fig. 2 and Fig. 3, for CRC and SHA, we can observe that the first clusters from the left are grouped exactly in the same way. For these workloads, after the profiling, we can arbitrarily select one execution flow from each cluster for a fault injection campaign and consider its failure distribution as a representative of the other member of that cluster. In this way, the variation in failure mode distribution of a workload can be discovered by performing fault injection campaigns on fewer number of execution flows.

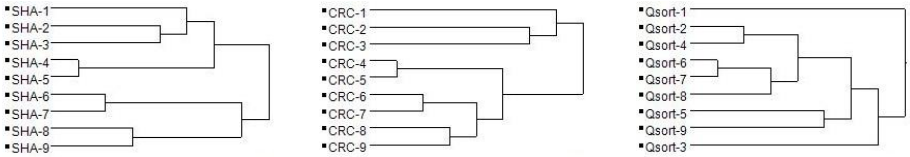

Fig. 2. SHA, CRC and Qsort clusters on assembly metrics

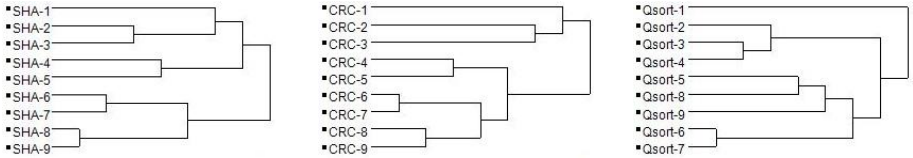

Fig. 3. SHA, CRC and Qsort clusters on the failure distributions

We quantify the reduction, *R*, of fault injection campaigns in equation (2).

$$R = (1 - C/I)*100 \quad (2)$$

Here *C* indicates the number of clusters at the first stage, and *I* is the total number of execution flows. For CRC and SHA, the reduction is 45%, which means that we can save about 45% of time. Hence, for these workloads we can profile their execution flows and on the basis of the obtained clusters decide whether to conduct a fault injection campaign or not. It is notable that input selection requires very limited human interactions and it is mostly accomplished by a fault-free run of the execution flow performed by Goofi-2, a signature extractor tool, and a data analysis tool. In our experimental environment, profiling costs up to 5 hours, while a fault injection campaign costs up to 2 days. This is a significant benefit of the proposed approach.

For Qsort there is no mapping between the clusters in the assembly space and the ones for the failure distribution. This might mean that for some applications like Qsort, where the failure distribution is dependent on more than just the length of input, other suitable assembly metrics are required. We exclude that this result is tied to

the choice of the clustering method since we also obtain identical results with other methods such as *average* and *ward* [12].

7 Related Work

Numerous works [14, 15, 16, 11] have assessed the effectiveness of hardware detection mechanisms in the presence of different fault models (such as pin level injection, stuck at byte, and bit flipping) while executing different workloads. In addition, an emerging research trend focuses on the implementation of software-implemented hardware fault tolerance mechanisms for detecting/correcting errors. Different implementation of software mechanisms at source level [2, 7] as well as at the assembly levels [3, 4, 17] has been assessed. These studies targeted a large variety of workloads and fault tolerance mechanisms without investigating their behavior to different inputs. In dependability benchmarking workloads are executed with realistic stimuli, i.e., inputs that come from the domain. In this area, the study [18] investigates the dependability of an automotive engine control system targeted with transient faults. The system under study is totally different from ours and no input selection approach is proposed. To the best of our knowledge, there is a little literature aiming to investigate the effects of transient faults on workload variations. In [5], matrix multiplication and selection sort are fed with three and two inputs, respectively. The fault model includes zero-a-byte, set-a-byte and two-bit compensation that differs from ours. Authors in [6] also estimated the error coverage for quicksort and shellsort, both executed with 24 different inputs. In addition, we study assembly level metrics with respect to the failure distribution (Section 6). While in performance benchmarking some study [19] explore the correlation between metrics and performance factors (e.g., power consumption), in the dependability field there is a no investigation on this area.

8 Conclusions and Future Work

We investigated the relationship between inputs of a set of workloads and the failure mode distribution. The experiments, carried out on an embedded system, demonstrate that for CRC and SHA, the length of input is linearly correlated to the percentage of value failure. Even though Qsort and BinInt do not show such a relationship, it is still notable that the input affects the failure distribution. Results illustrate that the percentage of faults detected by the hardware exceptions is workload dependent, i.e., it is not affected by the input. Additionally, a simple software-implemented hardware fault tolerant mechanism, TTR-FR, can successfully increase the coverage, on the average, to more than 97%, regardless of the input. As similar inputs (e.g., same length inputs) result in a similar failure distribution, we devised an approach to reduce the number of fault injections. Although the approach seems promising for workloads with a linear relation between the input property (e.g., length) and the failure distribution, additional metrics might be required for other workloads. Looking forward, we would like to improve the confidence in our findings by extending the study with other workloads, fault tolerance mechanisms, fault models and different compiler optimizations.

Acknowledgements. This work has partly been supported by VINNOVA-FFI BeSafe project and the FP7 European Project CRITICAL-STEP IAPP no. 230672.

References

1. Borkar, S.; "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10-16, 2005.
2. Rebaudengo, M.; Sonza Reorda, M.; Violante, M.; "A new approach to software-implemented fault tolerance," *Journal of Electronic Testing: Theory and Applications*, vol. 20, no.4, pp. 433-437, 2004.
3. Reis, G.A.; et. al.; "SWIFT: Software implemented fault tolerance," *Int. Symp. on Code generation and optimization (CGO'05)*, pp. 243-254, 2005.
4. Skarin, D.; Karlsson, J.; "Software implemented detection and recovery of soft errors in a brake-by-wire System," *7th European Dependable Computing Conf. (EDDC-07)*, pp. 145-154, 2008.
5. Segall, Z.; et al.; "FIAT-fault injection based automated testing environment," *18th Int. Symp. on Fault-Tolerant Computing (FTCS-18)*, pp. 102-107, 1988.
6. Folkesson, P.; Karlsson, J.; "Considering workload input variations in error coverage estimation," *3rd European Dependable Computing Conf. (EDDC-03)*, pp. 171-190, 1999.
7. Alexandersson, R.; Karlsson, J.; "Fault injection-based assessment of aspect-oriented implementation of fault tolerance," *41st Int. Dependable Systems & Networks Conf. (DSN)*, pp. 303-314, 2011.
8. Barbosa, R.; Vinter, J.; Folkesson, P.; Karlsson, J.; "Assembly-level pre-injection analysis for improving fault injection efficiency," *5th European Dependable Computing Conf. (EDDC'05)*, pp. 246-262, 2005.
9. Mibench Version 1, [Online] http://www.eecs.umich.edu/mibench/
10. Skarin, D.; Barbosa, R.; Karlsson, J.; "GOOFI-2: A tool for experimental dependability assessment," *40th Int. Dependable Systems & Networks Conf. (DSN)*, pp. 557-562, 2010.
11. Carreira, J.; Madeira, H.; Silva, J.G.; "Xception: A technique for the experimental evaluation of dependability in modern computer system," *IEEE Trans. Soft. Eng.*, vol. 24, no. 2, pp. 125-136, 1998.
12. Jain, A.; Murty, M.; Flynn, P.; "Data clustering: a review," *ACM Computing Surveys (CSUR)*, vol. 31, no. 3, pp. 264-323, 1999.
13. Natella, R.; Cotroneo, D.; Duraes, J.; Madeira, H.; "On fault representativeness of software fault injection", *IEEE Trans Soft Eng*, in press (PrePrint), 2011.
14. Kanawati, G.A.; Kanawati, N.A.; Abraham, J.A.; "FERRARI: a tool for the validation of system dependability properties," *22nd Int. Symp. on Fault-Tolerant Computing (FTCS-22)*, pp. 336-344 1992.
15. Madeira,H.; Rela, M.; Moreira, F.; Silva J.G; "RIFLE: A general purpose pin-level fault injector" *1st European Dependable Computing Conf. (EDDC-01)*, pp. 199-216, 1994.
16. Arlat, J.; et al..; "Comparison of physical and software-implemented fault injection techniques," *IEEE Trans. on Computers*, vol. 52, no. 9, pp. 1115-1133, 2003.
17. Martinez-Alvarez, A.; et. al; "Compiler-Directed soft error mitigation for embedded systems," *IEEE Trans. on Dependable and Secure Computing*, vol.9, no.2, pp. 159-172, 2012.
18. Ruiz, J.C.; Gil, P.; Yeste, P.; de Andrés, D.; Dependability benchmarking for computer systems. John Wiley & Sons, Inc., 2008, ch. "Dependability Benchmarking of automotive control system."
19. Eeckhout, L.; Sampson, J.; Calder, B.; "Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation," *IEEE Int. Workload Characterization Symp.*, pp. 2-12, 2005.