An Investigation of the Fault Sensitivity of Four Benchmark Workloads

Behrooz Sangchoolie
Department of Computer Science & Engineering, Chalmers University of Technology
behrooz.sangchoolie@chalmers.se

Fatemeh Ayatolahi
Department of Computer Science & Engineering, Chalmers University of Technology
fatemeh.ayatolahi@chalmers.se

Johan Karlsson
Department of Computer Science & Engineering, Chalmers University of Technology
Johan@chalmers.se

Abstract: This paper presents an experimental study of the fault sensitivity of four programs included in the MiBench test suit. We investigate their fault sensitivity with respect to hardware faults that manifest as single bit flips in main memory locations and instruction set architecture registers. To this end, we have conducted extensive fault injection experiments with two versions of each program, one basic version and one where the program is equipped with software-implemented hardware fault tolerance (SIHFT) through triple time redundant execution, majority voting and forward recovery (TTR-FR). The results show that TTR-FR achieves an error coverage between 94.6% and 99.2%, while the non-fault-tolerant versions achieve an error coverage between 55.8% and 81.1%. To gain understanding of the origin of the non-covered faults, we provide statistics on the fault sensitivity of different source code blocks, physical fault locations (instruction set architecture registers and main memory words) and different categories of machine instructions.

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

1 Introduction

The scaling of integrated circuit technology has led to remarkable increase in performance of computer systems. However, shrinking feature sizes have made transistors increasingly susceptible to a variety of failure mechanisms including process variations, wear-out effects, and ionizing particles [Bo05]. As a result, we can expect an increasing rate of transient, intermittent and permanent transistors faults in future integrated circuits. One way to prevent such faults from causing system failures is to make microprocessors and other integrated circuits internally fault tolerant. This can be achieved by the use of fault-tolerant latches, parity bits, error correction codes, lock-stepped processors, instruction retry mechanisms, and a variety of other techniques. However, these mechanisms can rarely achieve the level of error coverage required in safety- and mission-critical systems. In such systems, software-implemented hardware fault tolerance (SIHFT) provides an efficient and cost-effective approach for enhancing error coverage.

Software-implemented hardware fault tolerance has been widely studied in the literature over the last 30 years. Examples of techniques that are employed for achieving SIHFT include time redundant execution [Sc86, RSV04, NSV04], software-based control flow checking [OSM00, Mi92, LH07], algorithm-based fault tolerance [CD08], program encoding [WF07], and software assertions [An79]. There are also various ways of implementing SIHFT, such as by code transformation [Be00], manual programming [AK11], code interpreters [WF07], or aspect-oriented programming [AK11].

The effectiveness of SIHFT techniques is often evaluated by means of fault injection experiments. A common way to conduct such experiments is to measure the mechanism's ability to detect or mask bit errors in CPU registers and main memory locations [SK08]. In general, the error coverage achieved by SIHFT mechanisms varies for different programs. The error coverage also depends on the program inputs. Thus, it is essential to conduct fault injection experiments with different input sets [Se88, FK99, Di12] and different workloads.

This paper presents the results of extensive fault injection experiments with four workloads included in the MiBench test suit [Gu01]: *Secure Hash Algorithm*, *32-bit Cyclic Redundancy Check*, *Quick Sort Algorithm*, and *Binary String to Integer Convertor*. The main purpose of our experiments is to estimate the improvement in error coverage provided by a SIHFT technique called *triple-time redundant execution with majority voting and forward recovery*, or TTR-FR. To this end, we have injected single bit errors into CPU registers and main memory words of the MPC565 microcontroller.

The effectiveness of the TTR-FR mechanism has previously been evaluated for different target programs [AK11, Di12]. The authors of [Ul12] investigated the fault sensitivity of different code blocks, especially the voter, of a similar SIHFT mechanism utilizing majority voting.

This paper makes two main contributions. First, we evaluate the effectiveness of the TTR-FR technique for four programs. Second, we present an analysis of the origin of the non-covered faults. This analysis considers the fault sensitivity of different source code blocks, CPU-registers and main memory locations, and different categories of machine instructions. We believe that this analysis provides valuable insights for understanding the weaknesses of this SIHFT technique and thereby can help researchers develop more efficient SIHFT mechanisms.

The remainder of this paper is organized as follows. The target workloads are described in Section 2 and the TTR-FR mechanism in Section 3. The fault injection experimental setup is described in Section 4. In Section 5, we present the analysis of the extensive fault injections conducted on the workloads with/without the TTR-FR mechanism. Section 6 discusses related work. Section 7 provides conclusions and discusses future work.

2 Target Workloads

As already mentioned, we conducted our experiments with four workloads included in the MiBench suit [Gu01]. They are: i) secure hash algorithm (SHA), ii) 32-bit cyclic redundancy check (CRC), iii) quick sort algorithm (Qsort), and iv) binary string to integer convertor (BinInt). SHA is a cryptographic hash function which generates a 160-bit message digest. The SHA-1 algorithm, which we investigate in this paper, is applied in many security protocols and applications such as SSL, SSH and IPsec. The CRC workload is a software implementation of the well-known 32-bit cyclic redundancy check used in Ethernet and other communications protocols. The software version of CRC-32 is mainly intended for calculation of end-to-end checksums. Qsort is a recursive implementation of the well-known quick sort algorithm. This algorithm has been used as a workload in several previous fault injection experiments, e.g., [FK99, Ba05]. Finally, BinInt converts an ASCII binary string of 1s and 0s into its equivalent integer value.

Although we originally aimed at using the programs in the MiBench suite without any modifications, we decided for technical reasons to use slightly modified versions of the Qsort and SHA workloads. For quick sort, the MiBench implementation uses a built-in C function named *qsort* whose source code is not available, which prevented us from performing detailed analysis. For SHA, the MiBench implementation uses dynamic memory allocation, which is rarely used in embedded systems. Thus, we adopt another implementation of SHA[1] that uses static memory allocation.

The workloads were selected in a way to achieve diversity 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 contains one function with one loop, whereas SHA measures 125 LOC and has 6 functions.

We have conducted fault injection experiments with nine different inputs for each workload. We call the combination of an input and a workload an *execution flow*. Thus, for each workload, we performed experiments for 9 execution flows. The inputs for SHA and CRC execution flows consist of strings of varying sizes, from 0 to 99 characters. These inputs were 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, e.g., 20% of the elements are sorted in one input set while 40% of them are sorted in another input set. The inputs to BinInt consist of different random strings of 1s and 0s. Since an integer is a 32-bit data type, the length of the input string is limited to 32 characters.

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

3 Software-Implemented Hardware Fault Tolerance

The non-fault-tolerant version of the workloads consists of three major code blocks; startup, main function, and core function. The core function performs the main functionality of each workload. As an example, in Qsort, the sorting procedure is done in the Qsort's core function. The core function may contain several sub-functions, such as in SHA where the core function consists of six sub-functions.

In TTR-FR, the core function of the target workload is executed three times and the result of each run is compared with the other two runs using a software-implemented majority voter, see Figure 1. If only one run of the core function generates a different output, the output of the other two runs will be selected as the program output (majority voting). In case the workload is stateful, the state of the faulty run moves forward to a fault-free point (forward recovery). If none of the outputs match, then the voter signals an error.

4 Experimental Setup

Goofi-2 [SBK10], a fault injection tool developed in our research group, is used in our experiments. Faults are injected into the microcontroller via a Nexus debug interface. The workloads are executed on a Freescale MPC565 microcontroller, which uses the PowerPC architecture, see Figure 2.

Our fault model is the single bit flip model, which is also adopted in [CMS98, RSV04, Re05, SBK10]. Goofi-2 allows us to inject bit flip faults into instruction set architecture (ISA) registers and writable sections of the main memory (stack, data, etc.) of the microcontroller. The target registers are the ones used by the execution flows; general purpose registers, program counter register, link register, integer exception register, and condition register. We define a fault as time-location pair within an execution flow. The time then corresponds to the execution of a given machine instruction (i.e., a point in the execution flow), while the location is a randomly selected bit in memory word or CPU register read by the selected machine instruction. We use a pre-injection analysis feature [Ba05] provided in Goofi-2. The pre-injection analysis ensures that the faults are injected just before a register or a memory word is read by a machine instruction.

A *fault injection experiment* consists of injecting one fault and observing its impact on a workload. A *campaign* is a series of fault injection experiments with a given execution flow.

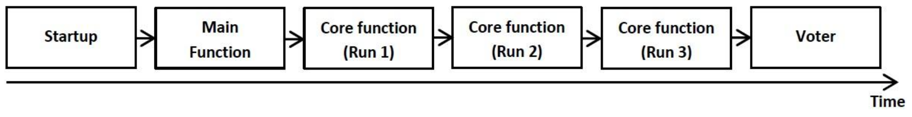

Figure 1. TTR-FR different code blocks

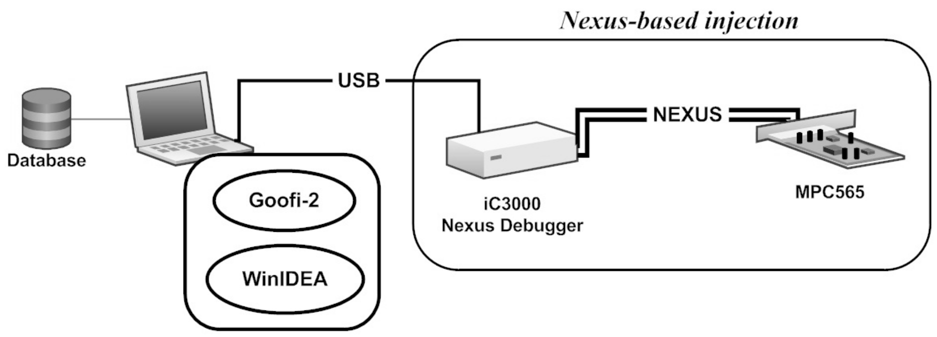

Figure 2. Experimental setup

5 Experimental Result

In this section, we present the results of fault injection campaigns conducted with the 4 workloads. We carried out 9 campaigns per workload which resulted in a total of 36 campaigns for the basic versions and 36 campaigns for the TTR-FR versions. Each campaign consists of 25000 experiments, except for campaigns conducted with CRC workload, which consist of 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 a 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.

We define the *error coverage* as the probability that an injected fault does not cause a value failure. The following equation estimates this probability:

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

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 experiments resulting in *no impact* and *timeout*. Experiments with no impact occur as a result of the internal robustness of the workload; hence

[2] We use a timeout value which is approximately 10 times larger than the execution time of the workload.

they contribute to the overall coverage of the system. Experiments that are classified as timeout are detected by Goofi-2. In a real application, we assume that a watchdog timer would detect these failures.

5.1 Results for Workloads without Software-Implemented Hardware Fault Tolerance

Table 1 shows the average failure distributions over the 9 execution flows of each workload. The 95% confidence interval for these measures varies from ±0.08% to ±0.89%. The highest percentage of covered errors (more than 80%) occurs in BinInt and Qsort, which are small workloads, while SHA, the largest and the most arithmetic-intensive workload, has an error coverage of around 55%.

Table 1. Failure distributions for workloads without TTR-FR (The values are average percentages over all execution flows).

Workload	VF	NI	DHW	TO	COV
CRC	31.18	23.38	44.38	1.06	**68.82**
SHA	44.18	14.59	39.73	1.50	**55.82**
Qsort	19.60	30.03	46.75	3.62	**80.40**
BinInt	18.92	36.16	41.93	2.99	**81.08**

The diagram in Figure 3 shows the average of the value failure distribution for the nine execution flows of each workload according to different instruction categories. Also shown in the diagram is the normalized average of the total number of executed instructions for each category. We see that the contribution of the load instructions is noticeably high for all workloads, while the contribution of the branch instructions is rather low for all workloads. These observations are mainly due to the fact that the number of executed load instructions is greater than the number of executed branch instructions in all workloads, and hence there are more fault locations for the load category compared to the branch category. The percentage of value failures originating from arithmetic instructions also varies between workloads depending on how arithmetic-intensive they are, e.g., in SHA around 33% of the value failures occur in the arithmetic category, while less than 5% of BinInt value failures are caused by arithmetic instructions.

We can see in Figure 3 that there are close correlations between the number of executed instructions and value failures of each instruction category. However, there are also cases where the percentage of value failure is higher than the percentage of executed instructions, such as in the load category of BinInt. In fact, such cases reveal that load and store instructions are more likely to cause value failures than other instructions in BinInt and CRC, respectively. In addition, if the percentage of value failures is lower than the percentage of executed instruction for a given category, this would mean that the faults in those categories are less sensitive, such as logical instructions in BinInt.

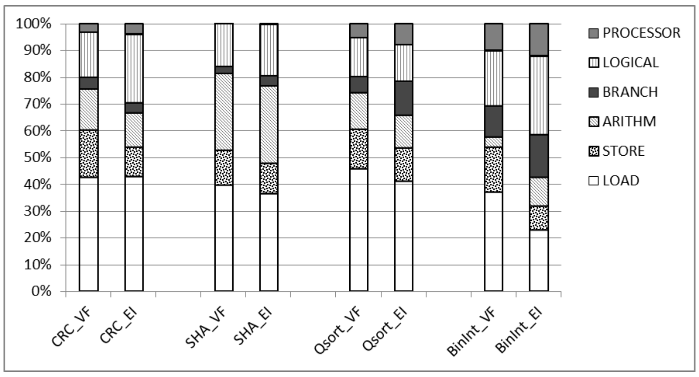

Figure 3. For each workload: *Left bar:* Value failure distribution over different instruction categories vs. *Right bar:* Percentage of the total number of executed instructions (All numbers are averages over all execution flows).

5.1.1 Value Failure Distribution over Different Code Blocks

In this section, we discuss the distribution of value failures in different code blocks of SHA. We choose to analyse the SHA program because it is the largest among our workloads. The aim of this analysis is to identify parts of the code that have high fault sensitivity. This information provides valuable insights for future research aimed at reducing the overhead of software-implemented hardware fault tolerance.

Table 2 shows the percentage of value failures in different code blocks of SHA. We can see that the percentage of value failure in different code blocks varies almost similarly in registers and memory locations. For example, the highest percentage of value failure occurs in *SHA1ProcessMessageBlock* while the lowest percentage of value failure corresponds to *STRlen* code block for both memory and register fault locations. As shown in Table 2, *SHA1ProcessMessageBlock* is responsible for 78% of the value failures in registers. Moreover, the total percentage of faults in this function is 74%. As Goofi-2 selects the fault locations randomly following a uniform distribution, it can be inferred that around 74% of the execution time is spent in this function. This indicates that the functions that are executed more often are also the main contributors to the lack of coverage.

Table 2. Value failure distribution in different code blocks of SHA

Code Blocks		% of Value Failures		% of faults
		Registers	Memory	
Core Functions	**Shareset**	0.13%	0.03%	0.17%
	Sha1result	0.10%	0.10%	0.12%
	Sha1input	19.49%	28.72%	22.56%
	SHA1ProcessMessageBlock	78.23%	69.39%	74.19%
	SHA1PadMessage	1.39%	1.76%	1.76%
	STRlen	0.05%	0.00%	0.63%
Main Function		0.61%	0.00%	0.57%

5.2 Results for Workloads with Software-Implemented Hardware Fault Tolerance

5.2.1 Failure Mode Distribution over Different Fault Locations

As mentioned in Section 4, our fault locations consist of instruction set architecture registers and writable main memory sections that are used by our workloads. The target registers are general purpose registers (GPR), program counter register (PCR), and miscellaneous group of registers (Misc) which consist of link register (LR), integer exception register (XER), and condition register (CR). Table 3 shows the average failure distributions over the 9 execution flows of each workload.

Table 3. Failure distributions for workloads extended with TTR-FR (The values are average percentages over all execution flows).

		Faults	*VF*	*NI*	*CSW*	*DSW*	*DHW*	*TO*	*COV*
CRC	GPR	44.6	0.58	23.77	44.22	0.11	30.30	1.02	**99.42**
	PCR	38.7	1.78	13.14	11.32	0.30	73.23	0.23	**98.22**
	Misc	3.3	0.44	52.11	40.40	0.02	6.73	0.30	**99.56**
	Memory	13.4	5.10	26.06	59.07	0.18	8.40	1.21	**94.9**
	Total	**100**	**1.65**	**20.77**	**33.43**	**0.19**	**43.22**	**0.73**	**98.35**
SHA	GPR	46.6	0.36	13.98	54.60	0.01	28.57	2.48	**99.64**
	PCR	37.8	1.47	13.96	16.59	0.41	66.52	1.04	**98.53**
	Misc	2.6	0.25	66.15	32.21	0.02	1.27	0.11	**99.75**
	Memory	13.0	0.33	11.06	82.80	0.00	4.05	1.76	**99.67**
	Total	**100**	**0.77**	**14.92**	**43.36**	**0.16**	**39.00**	**1.79**	**99.23**
Qsort	GPR	41.6	6.45	29.71	27.39	0.25	30.93	5.27	**93.55**
	PCR	37.5	4.03	17.68	6.20	1.31	70.31	0.48	**95.97**
	Misc	7.6	1.85	80.77	13.08	0.02	4.16	0.12	**98.15**
	Memory	13.4	8.17	27.33	42.37	1.32	17.68	3.13	**91.83**
	Total	**100**	**5.42**	**28.74**	**20.37**	**0.77**	**41.89**	**2.80**	**94.58**
BinInt	GPR	41.3	0.50	43.97	31.38	0.00	18.71	5.44	**99.5**
	PCR	41.8	2.73	17.48	4.27	0.20	74.78	0.53	**97.27**
	Misc	11.0	0.37	65.22	21.78	0.02	8.28	4.33	**99.63**
	Memory	6.0	33.79	33.79	52.95	0.00	12.17	0.00	**98.9**
	Total	**100**	**34.69**	**34.69**	**20.21**	**0.09**	**40.60**	**2.96**	**98.54**

We can see in Table 3 that even by using the TTR-FR, the coverage is not 100%. One of the main contributors to the lack of coverage is faults affecting the voter code block, which is not protected by any software-implemented hardware fault tolerance (more reasons on the lack of coverage are elaborated in Section 5.2.2). The highest coverage (more than 99%) is achieved for SHA, while Qsort has the lowest coverage (around 95%). The high coverage achieved for SHA is partly explained by the fact that a high proportion of injections (around 98%). was done in the core function of SHA. As the core block is protected by the TTR-FR, injected faults in this block have low probability of causing value failures.

Table 3 shows several similarities among the programs. For example, we can see that faults in the miscellaneous group of registers are always the ones that are least likely to cause value failures. They are also the ones that are least likely to be detected by hardware exceptions. There are also some notable differences among the programs. In CRC and Qsort, memory locations are the main contributors to the lack of coverage while in SHA and BinInt, the program counter register contributes the most to the lack of coverage. Another observation is that the faults in the program counter register always result in the highest percentage of *detected by hardware* and the lowest percentage *of corrected by software* experiments. This is mainly due to the fact that the injections in the program counter register cause control flow errors which are partially detected by the hardware exceptions. In addition, faults in the memory words contribute the most to the *corrected by software* category. Therefore, it can be concluded that even when the TTR-FR is being used, non-covered errors can significantly be decreased by protecting memory accesses and by providing additional control flow checks.

5.2.2 Value Failure Distribution over Different Code Blocks

In this section we investigate the distribution of value failures over different code blocks. Our analysis is performed on an execution flow of each workload which generates the highest percentage of value failure.

The core function and the voter are the main code blocks which cause the majority of non-covered errors. On the average, around 85% of the non-covered errors are in these two code blocks. Around 33% of these non-covered errors occurred in the voter and the remainder in the core function. This means that not only there still exist some faults in the core function that cannot be detected by TTR-FR, but also the voter itself is not totally immune to the faults. Also when we consider the total number of injections in each code block, the voter is more sensitive to faults than other code blocks since it is not protected by any software-implemented hardware fault tolerance mechanism.

In order to evaluate the behaviour of the sensitive code blocks (core function and voter), faults can be analyzed based on their locations (register categories and memory words). On the average, for the selected four execution flows, around 87% of the value failures are caused by the injections in registers, especially general purpose registers and program counter register. Sensitivity of these register categories is illustrated in Figure 4.

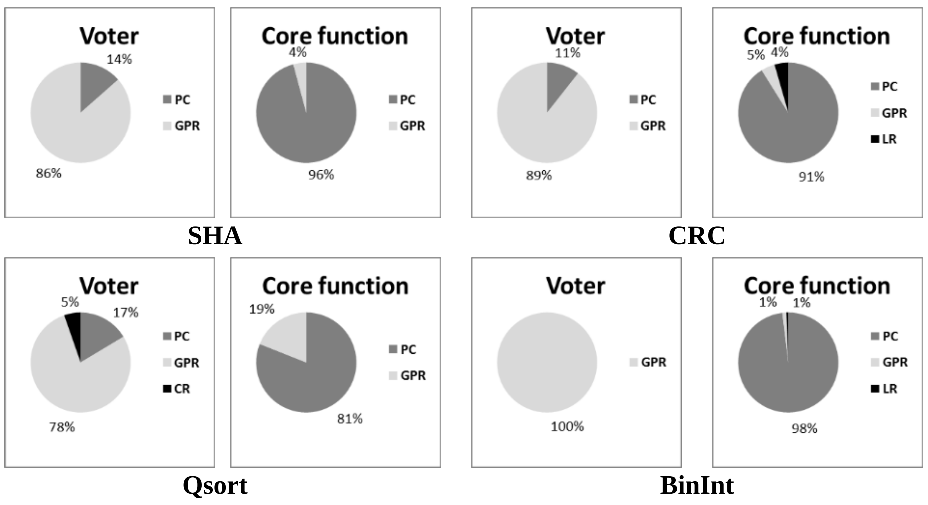

Figure 4. Value failure distribution in the voter and core blocks over different register categories. PC: Program Counter register, GPR: General Purpose Registers, LR: Link Register, CR: Condition Register.

Figure 4 shows the value failure distribution of the four execution flows over different register categories. It is notable that the general purpose registers are the main contributors to the lack of coverage in the voter code block. The control flow errors caused by the injections in the program counter register are detected by the hardware exceptions. This leaves the majority of the value failures in this block to correspond to general purpose registers.

Figure 4 also illustrates that the program counter register generates the highest percentage of value failure in the core function block. In fact, erroneous values caused by injections in the general purpose registers in each run of the core function are more likely to be masked by the other two runs. All faults in the program counter register, which cause the control flow of the program to jump outside the memory space of the program, are detected by the hardware exceptions. However, for other cases which caused value failures, we investigated the value inside the program counter register after each injection. In CRC, Qsort, and BinInt execution flows, the control of the program are moved to locations outside the core function where the voter is incorrectly executed or not executed at all. In SHA on the other hand, a few number of injections in the program counter register caused the control of the program to remain in the core function. In fact these few cases most probably caused the control flow to jump from one run of the core function to another run which eventually caused value failures.

6 Related Work

The effectiveness of different hardware detection mechanisms have been assessed in numerous works [Ka94, Ma94, Ar03]. These works targeted different fault models, such as pin level injection, stuck at byte, and bit flipping. In addition, different implementation of software fault tolerant mechanisms implemented at the source code level [RSV04, AK11] as well as at the assembly level [Re05, SK08, Ma12] has been assessed. These studies targeted a large variety of workloads and fault tolerance mechanisms. Only a few previous works have investigated the variation in error coverage for different workloads. In our previous work [Di12], we studied the error coverage of four workloads each running with 9 inputs. In [Se88], matrix multiplication and selection sort are fed with three and two inputs, respectively. The authors of [FK99] estimated the error coverage for quicksort and shellsort, both executed with 24 inputs.

7 Conclusions and Future Work

A previous investigation showed that TTR-FR achieved 95% error coverage for a break-by-wire application [AK11]. Our experiments shows that the Qsort workload achieved similar error coverage (94.5%), while the other three workloads, CRC, SHA, and BinInt achieved slightly higher error coverage. It is interesting to note that the most complex workload, SHA, achieved 99.2% error coverage. We also noticed that the main contributors to the lack of coverage in the voter code block are the faults in general purpose registers. Injections in the program counter register on the other hand caused the highest percentage of value failure in the core function. Overall our result showed that the TTR-FR needs to be complemented with other software-based mechanisms in order to achieve perfect or close to perfect error coverage. In our future work, we will aim to improve the confidence in our findings by extending the study with other workloads, fault tolerance mechanisms and fault models. We also plan to investigate how compiler optimization affects the error coverage of SIHFT techniques.

Acknowledgements. This work has partly been supported by VINNOVA-FFI BeSafe (Benchmarking of Functional Safety) project.

References

[AK11] 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.

[An79] Andrews, D.; "Using Executable Assertions for Testing and Fault Tolerance," Proc. of the 9th Int. Symp. on Fault-Tolerant Computing, pp. 102-105, 1979.

[Ar03] Arlat, J.; et al.; "Comparison of physical and software-implemented fault injection techniques," IEEE Trans. on Computers, vol. 52, no. 9, 2003.

[Ba05] Barbosa, R.; Vinter, J.; Folkesson, P.; Karlsson, J.; "Assembly-level pre-injection analysis for improving fault injection efficiency," 5th European Dependable Computing Conf., 2005.

[Be00] Benso, A.; Chiusano, S.; Prinetto, P.; Tagliaferro, L.; "A C/C++ SourcetoSource Compiler for Dependable Applications," Int. Conf. on Dependable Systems and Networks (DSN 2000), pp. 71-78, 2000.

[Bo05] 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.

[CD08] Zizhong Chen; Dongarra, J.; "Algorithm-Based Fault Tolerance for Fail-Stop Failures," IEEE Trans. on Parallel and Distributed Systems, vol. 19, no. 12, pp. 1628-1641, 2008

[CMS98] Carreira, J.; Madeira, H.; Silva, J.G.; "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computer System," IEEE Trans. on Soft. Eng., vol. 24, no. 2, pp. 125-136, 1998.

[Di12] Di Leo, D.; Ayatolahi, F.; Sangchoolie, B.; Karlsson, J.; Johansson, R.; "On the Impact of Hardware Faults - An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions," 31st Int. Conf. on Computer Safety, Reliability and Security (SafeComp 2012), 2012 (To appear).

[FK99] Folkesson, P.; Karlsson, J.; "Considering Workload Input Variations in Error Coverage Estimation," 3rd European Dependable Computing Conf. (EDDC-03), pp. 171-190, 1999.

[Gu01] Guthaus, M. R.; Ringenberg ,J. S.; Ernst, D.; Austin, T.M.; Mudge, T.; Brown, R. B.; "MiBench: A free, commercially representative embedded benchmark suite," IEEE 4th Annual Workshop on Workload Characterization, pp. 3-14, 2001.

[Ka94] Karlsson, J; et. al.; "Using heavy-ion radiation to validate fault-handling mechanisms," IEEE Micro, vol. 14, no. 1, pp. 8-23, 1994.

[LH07] Li, A.; Hong, B.; "Software implemented transient fault detection in space computer," Aerospace Science and Technology, vol. 11, no. 2-3, pp. 245-252, 2007.

[Ma12] Martinez-Alvarez, A.; et. all.; "Compiler-Directed Soft Error Mitigation for Embedded Systems," IEEE Trans. on Dependable and Secure Computing, vol.9, no.2, pp. 159-172, 2012.

[Ma94] 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.

[Mi92] Miremadi, G.; Karlsson, J.; Gunneflo, U.; Torin, J.; "Two software techniques for on-line error detection," 22nd Int. Symp. on Fault-Tolerant Computing (FTCS-22), pp. 328-335, 1992.

[NSV04] Nicolescu, B.; Savaria, Y.; Velazco, R.; "Software Detection Mechanisms Providing Full Coverage Against Single Bit-Flip Faults, " IEEE Trans. on Nuclear Science, vol. 51, no. 6, pp. 3510-3518, 2004.

[OSM00] Oh, N.; Shirvani, P.; McCluskey, E. J.; "Control-Flow Checking by Software Signatures," IEEE Trans. on Reliability, vol. 51, no. 2, pp. 111-122, 2002.

[Re05] Reis G.A.; et. al.; "SWIFT: Software Implemented Fault Tolerance," 3rd Int. Symp. on Code Generation and Optimization (CGO'05), pp. 243-254, 2005.

[RSV04] 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.

[SBK10] 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.

[Sc86] Schuette, M. A.; Shen, J. P.; Siewiorek, D. P.; Zhu, Y.; "Experimental evaluation of two concurrent error detection schemes, " 16th Int. Symp. on Fault-Tolerant Computing, pp. 138-143, 1986.

[Se88] Segall, Z.; et al.; "FIAT-fault injection based automated testing environment," 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18), pp. 102-107, 1988.

[SK08] 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.

[Ul12] Ulbrich, P.; Hoffmann, M.; Kapitza R.; Lohmann, D.; Schröder-Preikschat, W.; "Eliminating Single Points of Failure in Software-Based Redundancy," Pro. of the 9th European Dependable Computing Conf. (EDCC '12), 2012.

[WF07] Wappler, U.; Fetzer, C.; "Software encoded processing: Building dependable systems with commodity hardware," 26th Int. Conf. on Computer Safety, Reliability and Security (SafeComp 2007), pp. 356-369, 2007.