THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Understanding the Effects of Data Corruption on Application Behavior Based on Data Characteristics

Citation for published version:
Stefanakis, G, Nagarajan, V & Cintra, M 2015, Understanding the Effects of Data Corruption on Application Behavior Based on Data Characteristics. in *Computer Safety, Reliability, and Security: 34th International Conference, SAFECOMP 2015, Delft, The Netherlands, September 23-25, 2015, Proceedings.* Lecture Notes in Computer Science, vol. 9337, Springer International Publishing, pp. 151-165.
https://doi.org/10.1007/978-3-319-24255-2_12

Digital Object Identifier (DOI):
10.1007/978-3-319-24255-2_12

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Computer Safety, Reliability, and Security

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 24. Aug. 2022

Understanding the Effects of Data Corruption on Application Behavior Based on Data Characteristics

Georgios Stefanakis[1], Vijay Nagarajan[1], and Marcelo Cintra[2] *

[1] University of Edinburgh, United Kingdom
[2] Intel, Germany

Abstract. In this paper, the results of an experimental study on the error sensitivities of application data are presented. We develop a portable software-implemented fault-injection (SWIFI) tool that, on top of performing single-bit flip fault injections and capturing their effects on application behavior, is also data-level aware and tracks the corrupted application data to report their high-level characteristics (usage type, size, user, memory space location). After extensive testing of NPB-serial (7.8M fault injections), we are able to characterize the sensitivities of data based on their high-level characteristics. Moreover, we conclude that application data are error sensitive in parts; depending on their type, they have distinct and wide less-sensitive bit ranges either at the MSBs or LSBs. Among other uses, such gained insight could drive the development of sensitivity-aware protection mechanisms of application data.

1 Introduction

Reliability challenges have long been present in all parts of a system due to occurrences of anomalous physical conditions known as *hardware faults* [1]. Depending on the fault characteristics (location, type, timing, duration), the executing workload and the underlying hardware, faults can either (a) get masked by various levels of fault-masking effects (logic, architecture, application level) and result in a correct execution with no visible effects or (b) not get masked and result either in an observable execution upset (application crash, stall or delay) or an unobservable output corruption (Silent Data Corruption - SDC).

Motivated by the aforementioned fact, in this paper we study the effects of hardware-induced data corruption on application behavior in relation to the *high-level characteristics of the corrupted data* and the executing workload. Our purpose is to identify the error sensitivities and notice their variation for different data characteristics (usage type, size, user, memory space location) and also for different bit locations within the data; we define error sensitivity as the probability of a hardware fault in that data (or bit) to result in an SDC.

* This work is supported by EPSRC grant EP/M00113X/1 to the University of Edinburgh.

To do so, we employ software-implemented fault injection (SWIFI) to model transient single-bit faults in memory locations during application execution in an unprotected system and capture the corruption effect on the execution. Our focus is on gaining detailed error-sensitivity insight of the data accessed by an application. Therefore our SWIFI tool is *data-level aware*. Given an application binary, without need of its source code, our tool can finely control the location of the corruption in the application's memory space without further intruding the application behavior. Once a fault is injected at runtime, without need for binary file modifications per test, it tracks the corrupted data to classify them according to their use by the application. Meanwhile it monitors the execution's state and outcome to report back many diagnostics regarding the corruption characteristics/effects. As we monitor until completion, all fault-masking effects and corruption outcomes are captured.

SWIFI is commonly used in the literature mainly for system-level dependability assessment of reliability mechanisms [2–6]. Other works that study error sensitivities usually operate at a higher hardware level [7] or agnostic to the corrupted data characteristics [8] or to the exact corrupted bit. Instead, here we employ SWIFI to gain detailed error sensitivity insight at *application data level* and at a *per-bit granularity*. Thus, the main value of our study stems from the data-level awareness of the tests and the extensive set of tests performed. In general, there are many possible uses of the obtained error sensitivity insight. E.g., it can be used (a) to increase the protection of more sensitive data under SW-level fault-tolerant mechanisms, (b) to drive unequal protection of data words by assigning stronger protection to more sensitive bit ranges under HW-level fault-tolerant mechanisms, (c) to drive a sensitivity-aware SW-level modification of applications, (d) to reduce the testing space of dependability assessment, etc.

In this paper, we make the following main contributions:

(a) We establish a portable instrumentation-based SWIFI framework that can perform extensive tests on target binaries for a *data-level aware* study of the exact effects of data corruption on application behavior.

(b) After performing extensive (7.8M) fault-injection tests, we observe the error sensitivity of application data of the NPB-serial benchmarks based on the data characteristics, along with the variation among different bit locations of the data. We conclude that data are sensitive in parts; data holding output-related values have continuous less-sensitive bit ranges at their LSBs and memory addressing data at their MSBs. E.g., up to 32 LSBs of floating-point data in CG (Conjugate Gradient) each have <1% probability to cause an SDC if corrupted.

2 SWIFI Framework

In this section we present our instrumentation-based SWIFI framework that can perform extensive *data-level aware* fault injections and can track the corrupted data to report their high-level characteristics.

Our proposed **SWIFI framework** (Fig. 1) operates as follows: First, a fault-free run of the target binary is profiled to obtain (a) its expected correct output

(for SDC detection), (b) its normal execution time under instrumentation by our tool (for delayed/stalled execution detection) and (c) its total number of memory load accesses (for deciding the sample rate to drive the tests uniformly over the test space). Then a *single-fault injection tool* is repeatedly invoked on a clean instance of the target application, each time corrupting a different memory load access. Once all tests complete, the extensive reported results are aggregated to relate the corruption outcome to the corrupted data characteristics (Section 4).

The **single-fault injection tool** performs and monitors the fault-injection tests. In each test, just before a specified memory load access (*fault trigger*), a random bit of the accessed data is flipped to emulate a single-bit transient fault in the accessed memory location (*injected fault model*). Then the rest of the execution is *monitored* to *report* the exact end-to-end corruption effects. On top of that, the corrupted data are *tracked* to classify their high-level characteristics and, thus, report more corrupted data characteristics (Table 1). All these are performed by special software that emulates the behavior of expected hardware faults during application operation only (and not the kernel's).

Fault trigger: First, the application-under-test is instrumented until the execution reaches a specified memory load access to be corrupted. Using the memory load access as a fault trigger acts as both a spatial and a temporal trigger to invoke the injection routine just before the load operation. This trigger captures all possible times that a transient fault could occur and all possible live memory locations that could get corrupted. Thus it simplifies driving where/when to inject a fault by just selecting a load access, without relying on external events.

Fault injection and Fault model: Once the trigger is reached, the injection routine is invoked in a manner similar to a software trap. The chosen injected fault model emulates *single-bit transient faults in memory locations* by randomly flipping a bit of the data just before their access. The now-corrupted value is stored at the same memory location, without further intruding the application's original behavior, to avoid activating any reliability mechanisms of the system.

Due to our focus on data-level error sensitivity, the chosen fault model suffices without a need for precise realistic hardware fault models. Moreover, due to using instrumentation-based SWIFI, the fault model does not need to be adapted per target system but only to have the necessary high-level characteristics (type, duration, location). In particular, we chose a bit-flip *fault type* to ensure that data will always be corrupted at every test. The inject-before-load policy enforces

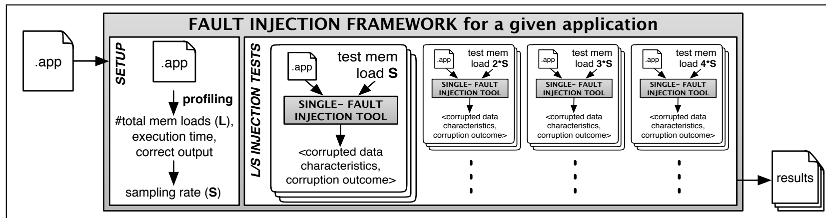


Fig. 1: Overview of our proposed data-level aware framework to capture the application behavior under corruption through extensive injection tests.

emulation of transient faults as the injected corruption will not persist after the corrupted location is overwritten. Modeling transient *fault duration* fits better our purposes as they affect only a single memory object. Finally, the injected *fault's location* is in main memory as a natural fit for a data-aware investigation.

The goal of our tool is mainly to observe how corruption in application data would affect the application's behavior. For our purposes corrupting data loaded from memory using an inject-before-load policy assists to cover as many as possible data used by an application while avoiding unnecessarily testing dead memory locations. This does not cover the entirety of application data; e.g. temporary data in registers that are never stored/loaded to/from memory or memory data never accessed. Despite faults are emulated only in memory, the fault model can translate to emulate faults occurring in other functional units too without revising the model and the injection policy. E.g., a fault in a register could be effectively emulated in a test where its value is stored in memory and then loaded but fault injected. Injecting directly faults all over the processor would be out of scope of this SW-implemented methodology.

Monitoring, Data tracking and Reporting: After the injection, the rest of the execution is still instrumented to monitor/report the effects of the corruption. As the instrumentation/analysis operates in a different virtual memory space to provide instrumentation transparency, the original binary observes the same addresses and values as it would in an uninstrumented execution [9]. Therefore, the original binary behavior is not changed, apart from the injected corruption, to ensure the non-intrusiveness of our injection tool.

Due to the chosen fault trigger, fault model and instrumentation-based injection we can perform data-level aware fault injection. Once the memory location is corrupted and loaded, we track it as an application variable to get its high-level characteristics. More precisely, at fault injection time, the tool tries to finely identify as many characteristics of the corrupted data as possible (Table 1). Attributes such as their location in the memory address space (global, heap or stack), size and user (system or user data) can be identified immediately.

Classifying the type of the corrupted data according to their use by the application (Fig. 2) can be either immediate or it may require tracking the data through the execution until a first meaningful use (i.e., to determine if they are used for addressing memory or not). To elaborate, at fault injection we can identify the first register (R) where the corrupted data (D') are stored. If it is

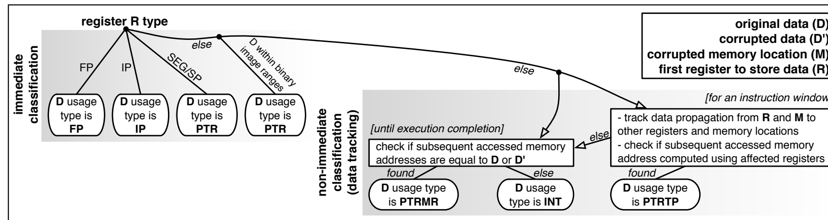


Fig. 2: Decision tree used by the single-fault injection tool to classify the corrupted application data according to their first use by the application.

Table 1: Reported corruption characteristics and corruption effects

Characteristics of corrupted data	
Injected bit-flip location, Memory address of corrupted data	
Memory space location: global, heap or stack	
Size:	1, 2, 4, 8 or 16 bytes
User:	System library data or application-space (user) data
Usage type:	**FP**: **F**loating-**P**oint data (immediate classification)
	IP: **I**nstruction **P**ointer (immediate classification)
	PTR: memory addressing data (immediate classification)
	PTRMR: mem. addressing data (classification by checking subsequent **M**em. **R**eferences)
	PTRTP: memory addressing data (classification by data tracking through **T**aint **P**ropagation until first use as memory addressing data within an instruction window)
	INT: **INT**eger data (if none of the above)

Corruption effects	
Total number of executed instructions	
Execution outcome:	**Correct output**
	Delayed correct output: when the total execution time is a set amount of times more than the normal uncorrupted execution time
	Application crash
	Application stall: due to excessive total executed instructions (or execution time)
	Silent data corruption (SDC): wrong output

an FP register, the instruction counter or a segment/stack pointer register, we can classify immediately the corrupted data as floating-point (FP), instruction pointer (IP) or memory addressing data (PTR) respectively.

If the data usage type cannot be determined immediately, data tracking and close monitoring of the execution is used. The corrupted (and the uncorrupted) value is checked against all subsequent accessed effective memory addresses to check if these values are used for memory addressing (PTRMR). Meanwhile, we use dynamic taint analysis [10] to track the data propagation from the first register (R) to hold the corrupted data and from the corrupted memory location (M). After every instruction, we track the corruption propagation to other registers and memory locations. This continues until a register whose contents have been affected by the original corruption is used for computing a memory address. If this happens within a specified instruction window, then the original corrupted data are reported as memory addressing data (PTRTP). If none of the above occur by the end of execution, the corrupted data are reported as INT.

We report memory addressing data as three separate categories, not only because they are identified by different means, but because they represent different usage cases. PTRs are memory addressing data that when loaded from memory have immediately the semantics of a pointer and are to be used as pointers. PTRTPs are memory addressing data that are identified through Taint Propagation analysis and are used to eventually compute a memory address; e.g. a loop counter that is used as memory offset. PTRMRs are identified by checking subsequent Memory References and are not immediately used as pointers.

After the injection, apart from the above, the tool keeps monitoring closely the execution to capture all possible corruption effects. The execution time is monitored to detect application stalls or delayed executions, the output is checked for correctness or SDCs, and fatal signals are caught to detect crashes. Once the execution completes (or stops due to a crash or stall), the tool reports back all the captured corruption characteristics and corruption effects (Table 1).

3 Experimental Setup

The proposed SWIFI framework was implemented as a set of scripts and dynamic binary instrumentation Pin tools [9]. Using Pin's instrumentation enabled the portability, transparency and efficiency properties of our tools. The full set of the ten workloads of the NAS Parallel Benchmarks [11] (64-bit, NPB-serial version 3.3.1, input class size S, gcc 4.4.6 -o3, Linux kernel 2.6.32) was extensively tested by our framework on a x86-64 computer cluster.

Before commencing with the individual fault-injection tests, the benchmarks were profiled (Table 2) to obtain their total memory load accesses and their normal uncorrupted execution time under instrumentation. The number of total memory loads indicated the test space size. Given that it ranged from 4.7M to 914.9M, summing up to a cumulative total of 2.28 billion, testing every bit of every memory load access would be impractical and unreasonable. Instead we set sample rates per benchmark (Table 2) to uniformly distribute our fault-injection tests over the test space of possible memory load accesses to corrupt. Moreover, in every test the bit-flip was randomly injected within the tested data to ensure an equal distribution of tested bits. The chosen sample rates ranged from 1/5 to 1/2947 to uniformly test each benchmark in approximately the same total time on the available computer cluster, where the embarrassingly parallel nature of the tests was exploited for a faster completion of the experiments (less than a week). The test space sampling brought the total number of performed fault-injection tests to 7.8M for the full benchmark suite (ranging from 310.4K to 1.33M for individual benchmarks). Despite the test space sampling, compared to related fault-injection based works, we performed significantly more extensive fault-injection tests that, coupled with the detailed collected test results, enabled us to thoroughly elaborate on them, as we discuss in the next section.

Table 2: Profiling information for the tested NPB-serial benchmarks

Bench-mark	Total memory loads (M)	Execution time (sec)	Sample rate	Memory loads tested (K)	Test space coverage (%)
BT	187.5	20.7	1/234	801.6	0.43
CG	111.9	22.7	1/153	731.3	0.65
DC	33.1	21.9	1/43	769.9	2.33
EP	778.2	52.3	1/2461	316.2	0.04
FT	112.0	31.9	1/216	518.5	0.46
IS	4.7	3.4	1/5	959.3	20.00
LU	62.9	16.5	1/62	1015.6	1.61
MG	10.6	13.7	1/8	1335.0	12.50
SP	66.9	15.3	1/62	1079.3	1.61
UA	914.9	53.3	1/2947	310.4	0.03
Total	2283.1	-	-	7837.6	0.34

4 Experimental Results - Discussion

In this section, we study the results to gain insight and elaborate on the varying error sensitivities of application data based on their high-level characteristics.

Application-level error-sensitivity variations: Fig. 3 shows the breakdown of the exact end-to-end corruption effects on the tested benchmarks. This

breakdown reconfirms that hardware faults have varying effects on application behavior. Out of the 7.8M performed fault-injection tests on NPB-serial, 61.1% resulted in correct execution. As for the rest outcomes, 23.5% of the total tests resulted in SDCs, 15% in application crashes, 0.3% in application stalls and less than 0.1% in delayed correct executions. More importantly the number of tests that corrupted silently the output varied per benchmark; the reported occurrences of SDCs ranged from 5.8% (DC) up to 37.9% (IS). This indicates that applications have different inherent error-sensitivity characteristics mostly attributed to their data-level sensitivity and their data access patterns.

Data-level error-sensitivity variations: Due to the data-level awareness of the fault-injection tests, we can study how the error sensitivity of application data varies in relation to their high-level characteristics. For this purpose we introduce the *experiment-based Data Vulnerability Factor* (eDVF) that we calculate using our testing results and we define as the statistical probability of a corruption in specified data categories to result in an SDC.

Fig. 4 shows the eDVF variation over the tested applications for the various high-level data characteristics that our fault-injection framework can identify. Generally most eDVFs are around 0.2, with limited exceptions going as high as 0.83, and quite a few being less than 0.05. This points that application data sensitivity can be characterized according to their characteristics. In a few cases, eDVFs are down to zero due to no reported SDC outcomes or due to absence of the particular data categories in the specific benchmark; in any case indicating them as less error-sensitive data for the application in question.

System library data (Fig. 4(c)) are less sensitive almost consistently across all benchmarks as they tend not to be output related and if corrupted tend to get masked or cause crashes. On the contrary, in some benchmarks, there is a trend of longer data being more vulnerable (Fig. 4(b)), as longer data often hold output-related values and thus if corrupted are more likely to corrupt the output too. As for the usage type eDVFs (Fig. 4(d)), there is no benchmark wide observation to be made. Despite that, they can be used in a per-application basis to rank the data sensitivities according to their type. Moreover, given the application's data access patterns, they can explain the total application error sensitivity.

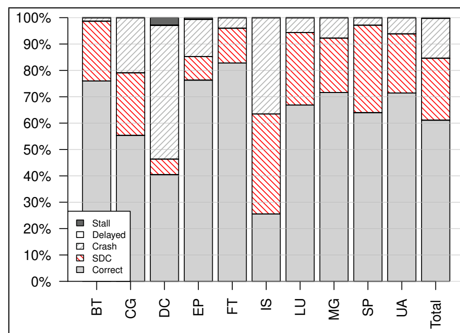
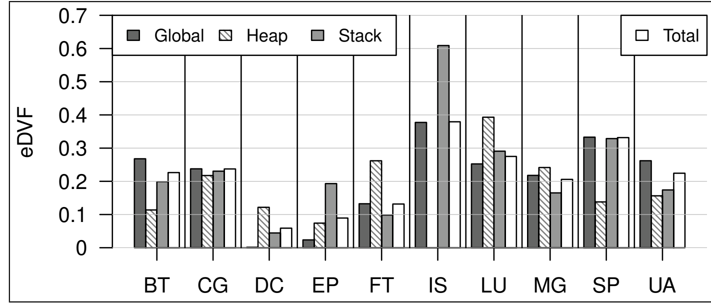
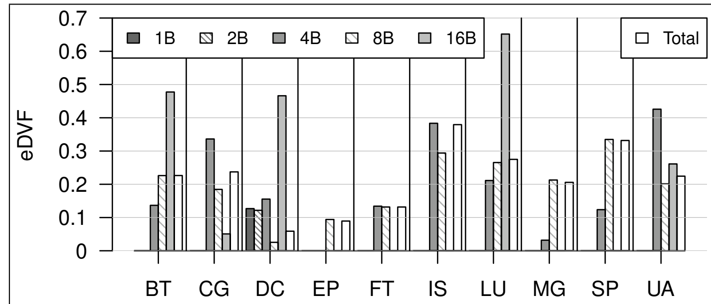


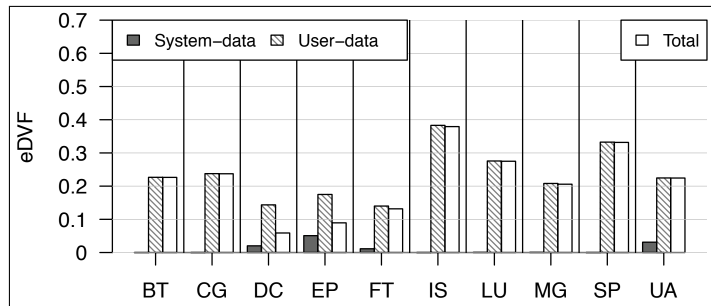
Fig. 3: Breakdown of corruption outcomes per tested NPB-serial benchmark and in total over all performed injection tests.



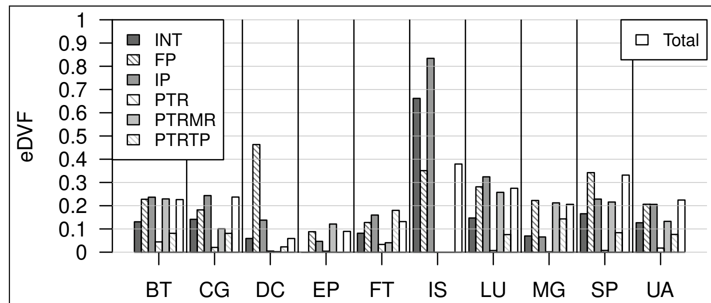
(a) Variations depending on **location of data** in the memory space



(b) Variations depending on **size** of data (in bytes)



(c) Variations depending on **user** of data (system library, user space)



(d) Variations depending on **usage type** of data

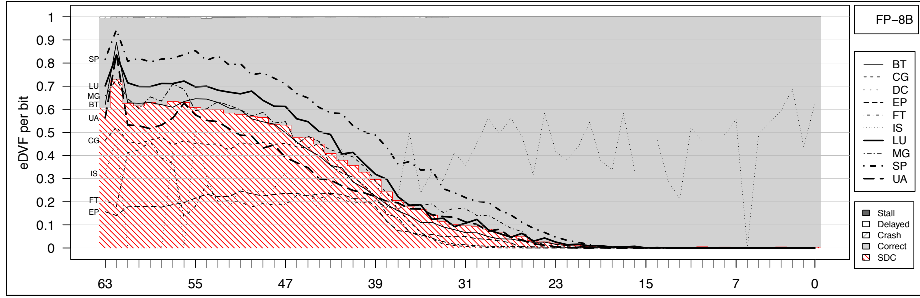
Fig. 4: Error-sensitivity variations of application data per tested NPB-serial benchmark depending on different high-level characteristics. *Total* indicates the reported SDCs for a given benchmark regardless of data characteristics.

Given the volume of our tests, our study captures the varying error sensitivities of application data per application with a high statistical confidence. As there are numerous combinations of data characteristics, only the variations per single data characteristics were presented here. Since the results of our experiments allow to compute the eDVFs for *combined* data characteristics, the application data error sensitivities can also be investigated more closely.

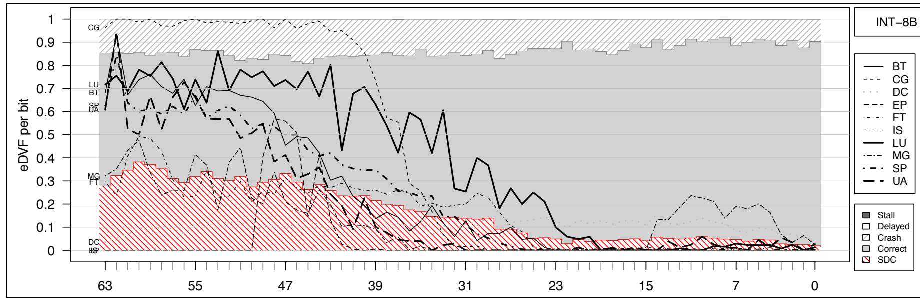
Per-bit data-level error-sensitivity variations: Moving down to per-bit investigation we can get more consistent error-sensitivity insight. Fig. 5-6 show how the per-bit error sensitivity varies among application data usage-type categories (see Table 1), while it shows common location patterns per each category across most of the tested applications. For given combinations of usage types and data sizes, the more-sensitive bits tend to concentrate in continuous bit ranges either at the MSBs (Fig. 5(a)-5(b)) or at the LSBs (Fig. 5(c)-6(b)) for most of the tested benchmarks, while the remaining bit ranges have generally near-zero eDVF per bit. All these suggest that we can clearly identify bit ranges within particular application data with distinct sensitivity levels to confidently conclude that application data are sensitive in parts.

More precisely, for the tested floating-point data (FP-8B, Fig. 5(a)) the less-sensitive bit ranges are located at their LSBs across most of the tested benchmarks, while moving towards the MSBs the per-bit eDVF increases steadily. When considering as less-sensitive bits those with per-bit eDVF less than 0.01, the less-sensitive bit-range width varies from 20 LSBs for SP up to 32 LSBs for CG, while many of these bits never resulted in an SDC. For the tested FPs, most of the non-SDC observed outcomes were correct executions. The observed sensitivity variations are explained by the nature of FPs where their LSBs only affect the accuracy of computations, are often discarded by rounding and tend not to affect the output. Moving to corruptions in MSBs it is expected that the data upset is intensified and as such the likelihood of resulting in an SDC increases. This also explains the varying less-sensitive bit-range width among applications as they have different precision requirements. Moreover it explains the different FP-sensitivity behavior in IS (Integer Sorting), where FPs are not part of the output but control the execution and are more likely to cause output corruption.

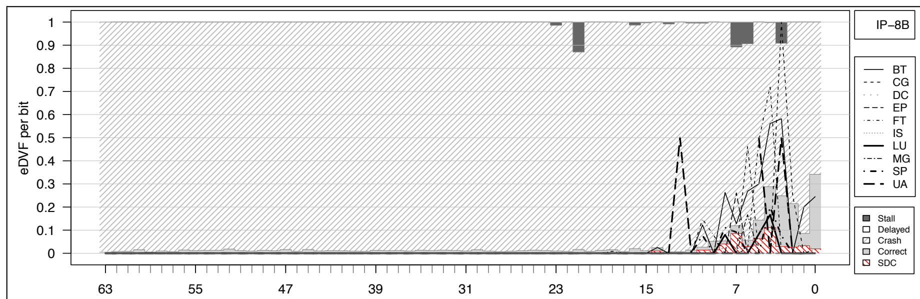
Similar to the FP data, the less-sensitive bit ranges in the tested INT data are located at their LSBs (INT-8B, Fig. 5(b)) but show higher eDVF per bit than their FP counterparts, while the pattern holds for higher eDVFs per bit when moving towards the MSBs. When considering as less-sensitive bits those with per-bit eDVF less than 0.10, the less-sensitive bit-range width varies from 24 LSBs for LU up to 43 LSBs for EP (not including DC and IS). This common behavior suggests that data holding values related to the computation, as both FPs and INTs do, tend to corrupt the output when they are corrupted at a greater magnitude (i.e., at MSBs). INTs can also be separated into distinct bit ranges with different sensitivity levels. Though, as they are used in many different application specific ways, there is more variation in the width of the less-sensitive ranges and not a common increasing eDVF pattern in the more-sensitive ones.



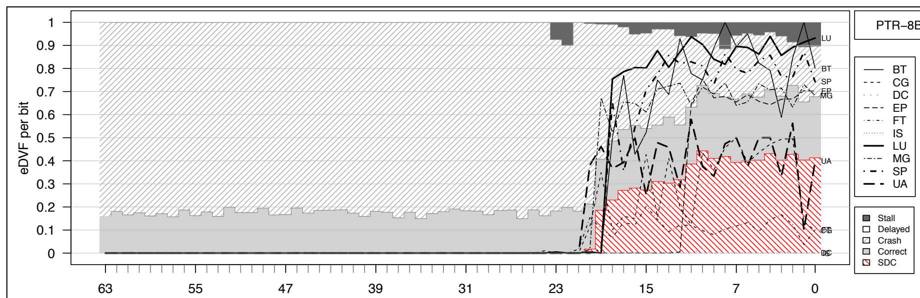
(a) Per-bit variations within **FP data (8 bytes)**



(b) Per-bit variations within **INT data (8 bytes)**

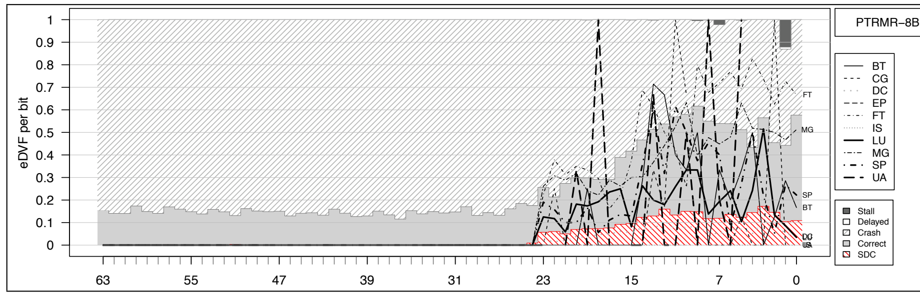


(c) Per-bit variations within **IP data (8 bytes)**

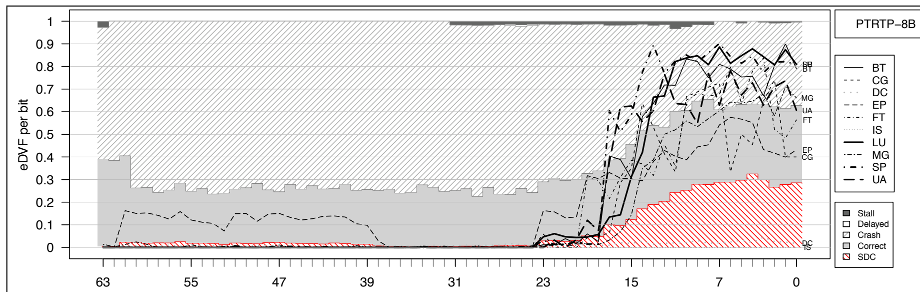


(d) Per-bit variations within **PTR data (8 bytes)**

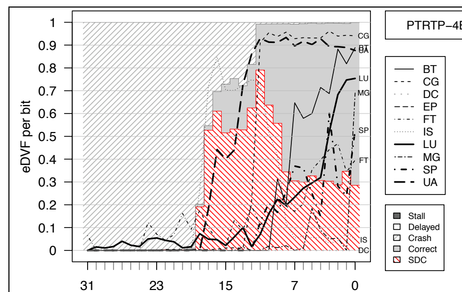
Fig. 5: Per-bit error-sensitivity variations (within combinations of data usage types and sizes) per tested NPB-serial benchmark. *Background bars* show the per-bit breakdown of corruption outcomes in total over all benchmarks.



(a) Per-bit variations within **PTRMR data (8 bytes)**



(b) Per-bit variations within **PTRTP data (8 bytes)**



(c) Per-bit variations within **PTRTP data (4 bytes)**

Fig. 6: Same as Fig. 5.

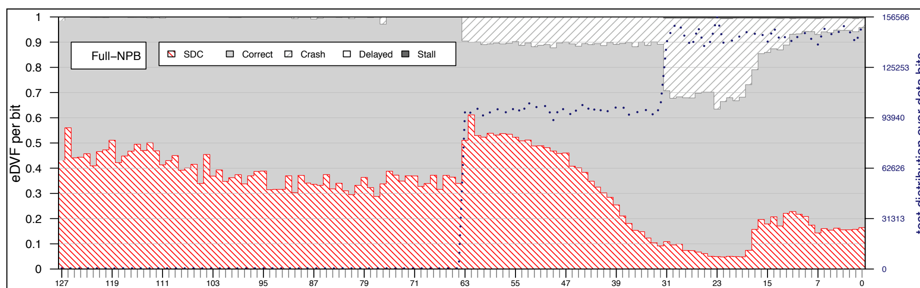


Fig. 7: Per-bit error-sensitivity variations (and breakdown of rest corruption outcomes) in total over all benchmarks, regardless of data usage type and size. *Dotted line* shows the number of times each bit was tested.

Moving on to memory addressing data (Fig. 5(c)-6(c)), we notice a reversal of the per-bit sensitivities; corruption at LSBs tends to result in SDCs and corruption at MSBs in application crashes mostly. This is because corruptions in MSBs of memory addressing data will lead to pointers into invalid memory locations and thus cause an application crash. On the contrary, corruptions in LSBs are more likely to lead to pointers into valid memory locations with undesired contents (or incorrect instructions) and thus corrupt the application output (or the instruction flow) but without causing an immediate crash. This is why the sensitive bit-range width of IP data is narrower than the PTR/PTRMR/PTRTP ones that are similar. The application program space is narrower compared to the data memory space and, thus, there are less bits in IPs (than in PTR/PTRMR/PTRTP) that if corrupted could still point to a valid location and not cause an application crash but an SDC. Nevertheless, the clear behavior for most benchmarks still enables to identify distinct sensitivity levels within different bit ranges of memory addressing data too.

As shown, it was promising to move into studying the effects of data corruption based on the corrupted bit location when considering specific high-level application data characteristics (i.e., usage type, size). Especially due to the usage-type-based classification, we were able to (a) further understand the previous eDVF usage-type results (Fig. 4(d)) and (b) get more consistent results among most benchmarks regarding the location patterns of more-sensitive application data parts. What changes across benchmarks is the less-sensitive bit-range width and the sensitivity intensity levels of the rest. For more detailed insight the eDVF per-bit variation can also be analyzed for *combined* data characteristics. As we can now identify clear bit ranges within particular application data with distinct sensitivity levels, the bit-level insight can be used instead of the higher-level eDVFs to characterize application data sensitivities more accurately.

Similar analysis could be performed for each of the other identified high-level characteristics of the corrupted data (i.e., location, user) or for the total per-bit eDVFs for all tested benchmarks combined (Fig. 7). Such analysis though does not provide the same clear insight, as the per usage type analysis, because the per-bit variation depends mostly on how the data-under-consideration are used.

5 Related Work

Various analytical and experimental techniques have been proposed usually for dependability assessment of systems and reliability mechanisms. Analytical techniques model the behavior of HW structures under the presence of faults usually through slow microarchitectural analysis [7]. An alternative is to use experimental techniques, such as experimental verification, error logging or full-system simulation under simulated faults. As these are also slow, the experimental approach of **fault injection** has been used instead to test real systems under realistic faults. Fault injection can be HW [12, 3, 4] or SW implemented [2–6, 13, 14]. In HW-implemented injection, faults are injected physically by electromagnetic interference and radiation [12] or through the circuit pins [3, 4]. Although

these inject real hardware faults that can reach all locations, they lack flexibility and are difficult to operate and control. Moreover, they suffer from low portability as they target specific systems, require special purpose dedicated hardware to access the tested hardware and may damage the tested system. On the contrary, **SW-implemented fault injection** (SWIFI) overcomes most of these drawbacks by injecting realistic faults using software methods. SWIFI achieves higher properties of repeatability, controllability (in space and time), reproducibility, non-intrusiveness and efficacy [12]. Generally, in SWIFI, transient faults are injected by adding traps or replacing instructions, either at pre-runtime or at runtime. Pre-runtime injection methods mutate the application, i.e., by substituting instructions and program data [14] or by source code mutation [13]. Runtime injection methods most commonly corrupt memory or register contents using time-based, path-based or stress-based triggers, while they inject faults by direct program memory image corruption [6], dynamic process control structure corruption using debugging registers [2, 4, 5], forcing execution of pre-loaded routines using software traps [3, 13] or hardware breakpoints [3, 4].

Varying error sensitivities: Many approaches have implied a sensitivity classification of hardware parts albeit without a formal exploration [15, 16]. Similarly, other approaches implied a data sensitivity classification based on cache access patterns [17]. Moving to higher abstraction levels is promising to observe the varying effects of faults [4, 13]. Similar variations are implied in approaches where code segments [18, 19] or individual instructions [20] are marked as critical. More formally, a reliability-aware analysis can be used to detect statistically-vulnerable code segments [21] and instructions [20, 22]. The same applies for data-level sensitivities, where it has been implied that not all data are equally sensitive, i.e., by marking data segments as non-critical if they only affect the output of multimedia workloads [23], or as approximate if their preciseness is not required [24]. More formally, analysis can further elaborate on the criticality of data, e.g., by profiling data according to their liveliness [22], or by detecting sensitive bit ranges [25] within data with known value ranges. For the same purpose, finely-controlled fault injection and execution monitoring can be used for more insight on the corruption effects in a per-bit manner [8].

6 Conclusion

In this paper we developed an instrumentation-based SWIFI tool that is data-level aware and tracks application data in order to gain detailed error-sensitivity insight at application data level. We showed through a set of extensive fault-injection experiments on NPB-serial that we can analyze the exact effects of data corruption on application behavior based on the high-level characteristics of the corrupted data (usage type, size, user, memory space location). This not only enabled to capture the varying sensitivities of data given their characteristics but also to identify less-sensitive bit ranges within data. Among many potential future uses, the gained insight could motivate the development of sensitivity-aware protection mechanisms trading-off between protection cost and fault coverage.

References

1. Sorin, D.J.: Fault tolerant computer architecture. Synthesis Lectures on Computer Architecture **4**(1) (2009)
2. Kanawati, G., et al.: FERRARI: A flexible software-based fault and error injection system. Trans. on Computers **44**(2) (1995)
3. Skarin, D., et al.: GOOFI-2: A tool for experimental dependability assessment. In: DSN. (2010)
4. Stott, D., et al.: NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. In: IPDS. (2000)
5. Carreira, J., et al.: Xception: A technique for the experimental evaluation of dependability in modern computers. Trans. on Software Engineering **24**(2) (1998)
6. Segall, Z., et al.: FIAT - Fault injection based automated testing environment. In: FTCS. (1988)
7. Mukherjee, S.S., et al.: A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: MICRO. (2003)
8. Ayatolahi, F., et al.: A study of the impact of single bit-flip and double bit-flip errors on program execution. In: SAFECOMP. (2013)
9. Luk, C.K., et al.: Pin: Building customized program analysis tools with dynamic instrumentation. In: PLDI. (2005)
10. Zhu, Y., et al.: Privacy Scope: A precise information flow tracking system for finding application leaks. Technical report (2009)
11. Bailey, D., et al.: The NAS parallel benchmarks. Intern. Journal of High Performance Computing Applications **5**(3) (1991)
12. Arlat, J., et al.: Comparison of physical and software-implemented fault injection techniques. Trans. on Computers **52**(9) (2003)
13. Hiller, M., et al.: PROPANE: An environment for examining the propagation of errors in software. In: ISSTA. (2002)
14. Gerardin, J.P.: The DEF.Injecto test instrument, assistance in the design of reliable and safe systems. Computers in Industry **11**(4) (1989)
15. Greskamp, B., et al.: BlueShift: Designing processors for timing speculation from the ground up. In: HPCA. (2009)
16. Ernst, D., et al.: Razor: A low-power pipeline based on circuit-level timing speculation. In: MICRO. (2003)
17. Zhang, W., et al.: Performance, energy, and reliability tradeoffs in replicating hot cache lines. In: CASES. (2003)
18. de Kruijf, M., et al.: Relax: An architectural framework for software recovery of hardware faults. In: ISCA. (2010)
19. Reis, G.A., et al.: SWIFT: Software implemented fault tolerance. In: CGO. (2005)
20. Borodin, D., et al.: Instruction-level fault tolerance configurability. Journal of Signal Processing Systems (2009)
21. Feng, S., et al.: Shoestring: Probabilistic soft error reliability on the cheap. In: ASPLOS. (2010)
22. Mehrara, M., Austin, T.: Exploiting selective placement for low-cost memory protection. TACO **5**(3) (2008)
23. Lee, K., et al.: Partially protected caches to reduce failures due to soft errors in multimedia applications. Trans. on VLSI Systems **17**(9) (2009)
24. Sampson, A., et al.: EnerJ: Approximate data types for safe and general low-power computation. In: PLDI. (2011)
25. Chang, J., et al.: Automatic instruction-level software-only recovery. In: DSN. (2006)