# LLFI: An Intermediate Code Level Fault Injector For Soft Computing Applications

Anna Thomas and Karthik Pattabiraman
Department of Electrical and Computer Engineering, University of British Columbia
{annat, karthikp}@ece.ubc.ca

***Abstract*—Hardware errors are on the rise with reducing chip sizes. However, a certain class of applications called soft computing applications, (e.g., multimedia applications) can tolerate most hardware errors, except those that result in outcomes that deviate significantly from the error-free outcomes. We term such outcomes as Egregious Data Corruptions (EDCs).**

**To identify source code level characteristics of EDC causing faults, we built an LLVM based fault injector tool called LLFI. LLFI performs fault injection at the intermediate code level of the application. We quantitatively validate LLFI accuracy with respect to assembly level fault injection. Using LLFI, we performed a study to identify the correlation between faults in specific data types, and EDC outcomes. This data categorization will help us identify detector placement locations with high coverage for EDC causing faults.**

## I. Introduction

With the reduction of chip sizes and the concomitant increase in the number of transistors on a chip, the frequency of hardware faults is on the rise. Traditionally, hardware errors have been tolerated through hardware redundancy or guard banding. Unfortunately, hardware-only solutions have high energy overheads, and become untenable as power consumption becomes a dominant concern in processor design [1].

Recently, there have been several proposals to selectively expose hardware faults to the software layer and tolerate them [2], [3], [4], [5], [6]. These proposals leverage the ability of certain software applications to tolerate faults in their data, and still produce acceptable outputs. Such applications are called soft computing applications [7].

Examples of soft computing applications are multimedia decoding applications, which can tolerate blurry decoded images, and machine learning applications, which can tolerate noise. These applications have an associated fidelity metric, which is a quantitative measure of the output quality. For example, in the case of image and video decoders, the fidelity metric is peak signal-to-noise ratio (PSNR). As long as the produced output quality does not deviate significantly from the fidelity metric, it is deemed acceptable. We use the term *Egregious Data Corruptions (EDCs)* to denote outcomes whose quality deviates significantly from the fidelity metric, i.e., unacceptable outcomes.

Our goal is to identify source level heuristics that help identify optimal locations for high coverage detectors for EDC causing faults. We address this goal by performing fault injection experiments, to identify data categories that contribute to causing an EDC. Although fault injections have been usually performed at the assembly level, it is difficult to map the fault behaviour and propagation back to the source level. This mapping problem can be avoided by injecting the fault at the source code level, but doing so does not accurately model hardware faults. This is because many hardware faults at the lower layers of the system stack get masked and are not visible at the application layer. Also, faults that affect certain control flow instructions and special purpose registers cannot be simulated at the source code level.

In this work, we build a fault injection tool LLFI, that injects faults into the Low Level Virtual Machine (LLVM) [8] framework's *intermediate code* of the application. LLVM is a collection of reusable compiler tools and components, and allows analysis and optimization of code written in multiple languages, including C/C++. The intermediate code of LLVM is at a higher level than assembly code and has support for types. However, it encodes more information than source code, such as address computations of loads and stores, and hence injection is more accurate than at the source level.

We make the following contributions in this work:

1) We build LLFI, a fault injection mechanism at the LLVM framework's intermediate code level. LLFI allows fault-injections to be performed at specific program points and into specific data types, and compare the results.
2) We quantitatively compare the fault injection results at the LLVM intermediate code level versus those performed directly at the machine code level using a PIN-tool based fault injector [9].
3) Using LLFI, we perform fault injection into soft computing applications, and distinguish EDCs from Silent Data Corruptions (SDCs). We identify certain data categories, i.e., control and pointer data, and extract the correlation between these data categories and EDC causing faults. We study this correlation by monitoring these data categories, using our tracing mechanism in LLFI.

**Related Work:** Schiffel et al. [10] developed an LLVM based fault injector tool. They consider different kinds of faults such as perturbing the instruction, its operands, and its result. However, the tool is not built for soft computing applications, and it is not clear what user input is required to choose the location for fault injection. Further, they do not validate their results. Other work has investigated the resilience of applications at the assembly code level [11]. However, this makes it difficult to formulate source-level heuristics, which

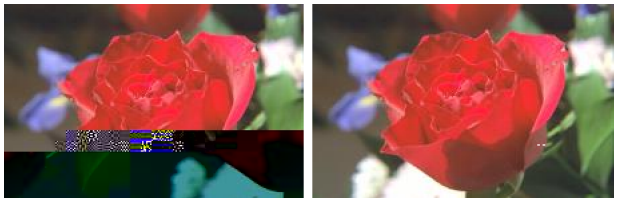

*Fig. 1:* The EDC causing fault decoded image (left) versus Non-EDC causing fault decoded image (right) from the JPEG decoder

is our goal.

## II. Background

**Egregious Data Corruptions (EDCs)** are application outcomes that deviate significantly from the fault free outcome, i.e., they affect outputs egregiously. This deviation is quantified by a fidelity metric that is well defined for most soft computing applications [12]. Silent Data Corruptions (SDCs), or outcomes that result in any deviation in the output from the fault free outcome, are a superset of EDCs. An SDC is classified as EDC or Non-EDC, depending on the fidelity threshold value of the outcome. Non-EDCs are the SDCs with small deviation in output, relative to the threshold.

EDC is a relative term as it depends on how the user sets the fidelity threshold. In this paper, we focus on detecting errors under the assumption that the user tolerates most small deviations in outputs, i.e., the application is used under relaxed conditions. For example, in image and video decoding applications, we set the fidelity threshold based on whether the frames are corrupted to the point of being unrecognizable by a human, or are of very poor image quality.

The example in figure 1 shows the faulty decoded images of the JPEG decoder (part of Mediabench [13]), when a fault is injected into the program. The fidelity threshold is the PSNR between the fault-free decoded image, and the faulty decoded image. As the PSNR value becomes lower, the output corruption becomes more egregious. Assuming a fidelity threshold value of PSNR 30, the faulty image on the left with a PSNR of 11.37 is classified as an EDC, while the faulty image on the right with a PSNR value of 44.79 is classified as a Non-EDC. The comparison is performed with respect to the base image, which we do not show.

## III. Fault Injection

In this section, we present our fault injection tool which operates at the LLVM Intermediate Representation (IR) level. We first present the fault model, followed by the design of our tool. We finally discuss the accuracy of our tool versus assembly level fault injection.

### A. Fault Model

We consider transient hardware faults that occur in the processor. These are usually caused by cosmic ray or alpha particle strikes affecting flip flops and logic elements. We consider faults that occur in the functional units, i.e., the ALU and the address computation for loads and stores. However, faults in the memory components such as caches are not considered, since these components are usually protected at the architectural level using ECC or parity. We do not consider faults in the control logic of the processor as this is a small portion of the processor area, nor do we consider faults in the instructions, as these can be handled through control-flow checking techniques [14].

### B. LLVM Fault Injector: LLFI

We developed LLFI, a program level fault-injection tool for performing the fault injection experiments. LLFI works at the LLVM compiler's intermediate level, and enables tracing the propagation of the fault in the program by instrumenting the program at selected points. LLFI is closely integrated with the LLVM compiler, and can hence support a wide variety of programs and programming languages [8].

LLVM uses the Static Single Assignment (SSA) form [15]. SSA requires a variable be assigned exactly once in the program i.e., every variable in the program has a unique instruction that assigns to it. We chose LLVM because:

1) Its intermediate code is a typed language, in which source-level constructs can be easily represented. In particular, it preserves the variable and function names, making source mapping feasible.
2) It has support for program analysis and transformations which makes it easier to study the effect of fault injection at a higher level than the assembly language, while still accounting for details not visible in the source code, such as address computation.
3) The lowering of the intermediate code to specific architectures is robust [16] - the analysis done at the IR level can be used at assembly level deterministically and with reasonable accuracy.

LLFI injects a fault, i.e., a single bit flip into the destination register of exactly one dynamic instance of an instruction chosen at random (among all the executed instructions), and classifies the outcome of the fault by comparing the final outcome with the fault free outcome. The fault-free or baseline outcome is obtained by running the original executable with the same input, but without any injected faults. The faulty outcomes are classified into Crash, Benign, EDCs and Non-EDCs. The EDCs are separated from the Non-EDCs based on the fidelity threshold value.

**Example:** We explain the fault injection process using the factorial program in Figure 2 as an example. The original IR for the corresponding C code is shown in Figure 3. The value of `n` (the number whose factorial is calculated) is assigned in line 7 of the C code. This corresponds to %2 in line 6 of Figure 3. The basic block `bb1` in lines 15-18 in the IR corresponds to the loop header at line 9 in the C code. The `phi` [1] node at line 15 gets the value of the iterator variable `i`, which is initialized to 1 (from basic block `entry`) or obtained from the incremented value within the loop at basic block `bb`. Similarly, `fact` is also assigned the `phi` node value where it

[1]Phi is a construct used in SSA form

is either 1 (the initial value), or the value of `fact` from within the loop at line 10.

Figure 4 shows the original IR statically instrumented for fault injection. Each instruction with a return value has a call to the fault injection function `injectFault`. For example, lines 10 and 11 in Figure 3 have corresponding fault injection calls at lines 17 and 19 in the instrumented IR in Figure 4. The arguments of the `injectFault` function are the static fault ID, the type of call (fault injection call type has value 0), and the result of the instruction itself. This function flips a randomly chosen bit in the instruction's result, and returns this faulty result. Next, all `uses` of the original instruction are replaced by the fault injected instruction. For example, line 15 in the original IR corresponds to line 23 in the instrumented IR, and %4 is replaced by the corresponding fault injection call %fi7. Since LLVM is a typed IR, we handle instructions of different types by having separate fault injection calls for each type.

At runtime, we first obtain the total number of dynamic instructions by running the original factorial IR. Second, for each fault injection run, we choose a random instruction instance from the set of all dynamic instructions for fault injection. This is done by the function `initInjections` inserted at the beginning of the program (at line 3 in Figure 4). Third, for each instruction, the `injectFault` function checks if the particular instance is the dynamic instance to be fault injected, and if so, flips a single bit in the instruction result. This fault-injected result is propagated to all the `uses` of this instruction in LLVM.

### C. Accuracy of LLFI

While the LLVM intermediate code is close to the assembly code, it does not correspond one-to-one with the assembly language. We qualitatively assess the correspondence between the LLVM intermediate code and the assembly code for fault-injection purposes. The differences are presented in table I. We quantify the effect of these differences in Section VI-B.

## IV. Data Tracing

One of the main goals of this work is to identify data variables in the program at which error detectors must be placed to avoid EDCs. To determine these variables, we trace the propagation of faults injected in the program using LLFI, and monitor the values of certain variables in the program. Our goal is to determine if the values of the data items exhibit a deviation from their fault-free values. To achieve this, we first run the program without injecting any faults and collect the values of the data items (obtained by instrumenting the program with the LLVM compiler as explained in Section V). We then compare the value of the data items between the fault-free and fault-injected runs.

We split the results according to the type of the monitored data items, since we wanted to see if there was a correlation between the fault outcome and the type of data that exhibits a deviation from its fault-free value. Because we perform the injection and analysis at the LLVM intermediate-code level, we can track the detailed provenance of the data and its

```c
#include<stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
  int i,fact, n;
  n = atoi(argv[1]);
  fact = 1;
  for(i=1;i<=n;i++)
  {
    fact = fact * i;
  }
  printf("%d\n",fact);
}
```

*Fig. 2:* C code for factorial program

```
1  define i32 @main(i32 %argc, i8** %argv) nounwind {
2  entry:
3    %"alloca point" = bitcast i32 0 to i32
4    %0 = getelementptr inbounds i8** %argv, i64 1
5    %1 = load i8** %0, align 1
6    %2 = call i32 (...)* @atoi(i8* %1) nounwind
7    br label %bb1
8
9  bb:                                               ; preds = %bb1
10   %3 = mul nsw i32 %fact.0, %i.0
11   %4 = add nsw i32 %i.0, 1
12   br label %bb1
13
14 bb1:                                              ; preds = %bb, %entry
15   %i.0 = phi i32 [ 1, %entry ], [ %4, %bb ]
16   %fact.0 = phi i32 [ 1, %entry ], [ %3, %bb ]
17   %5 = icmp sle i32 %i.0, %2
18   br i1 %5, label %bb, label %bb2
19
20 bb2:                                              ; preds = %bb1
21   %6 = call i32 (i8*, ...)* @printf(i8* noalias getelementptr
22        inbounds ([4 x i8]* @.str, i64 0, i64 0), i32 %fact.0) nounwind
23   br label %return
24
25 return:                                           ; preds = %bb2
26   ret i32 undef
27 }
```

*Fig. 3:* LLVM IR for the factorial program

```
1  define i32 @main(i32 %argc, i8** %argv) nounwind {
2  entry:
3    call void @initInjections(i8* getelementptr
4           inbounds ([21 x i8]* @NameStr, i32 0, i32 0))
5    %"alloca point" = bitcast i32 0 to i32
6    %fi2 = call i32 @injectFault0(i32 2, i32 0, i32 %"alloca point")
7    %0 = getelementptr inbounds i8** %argv, i64 1
8    %fi3 = call i8** @injectFault1(i32 3, i32 0, i8** %0)
9    %1 = load i8** %fi3, align 1
10   %fi4 = call i8* @injectFault2(i32 4, i32 0, i8* %1)
11   %2 = call i32 (...)* @atoi(i8* %fi4) nounwind
12   %fi5 = call i32 @injectFault0(i32 5, i32 0, i32 %2)
13   br label %bb1
14
15 bb:                                               ; preds = %bb1
16   %3 = mul nsw i32 %fi8, %fi1
17   %fi6 = call i32 @injectFault0(i32 6, i32 0, i32 %3)
18   %4 = add nsw i32 %fi1, 1
19   %fi7 = call i32 @injectFault0(i32 7, i32 0, i32 %4)
20   br label %bb1
21
22 bb1:                                              ; preds = %bb, %entry
23   %i.0 = phi i32 [ 1, %entry ], [ %fi7, %bb ]
24   %fact.0 = phi i32 [ 1, %entry ], [ %fi6, %bb ]
25   %fi8 = call i32 @injectFault0(i32 8, i32 0, i32 %fact.0)
26   %fi1 = call i32 @injectFault0(i32 1, i32 0, i32 %i.0)
27   %5 = icmp sle i32 %fi1, %fi5
28   %fi9 = call i1 @injectFault3(i32 9, i32 0, i1 %5)
29   br i1 %fi9, label %bb, label %bb2
30
31 bb2:                                              ; preds = %bb1
32   %6 = call i32 (i8*, ...)* @printf(i8* noalias getelementptr
33          inbounds ([4 x i8]* @.str, i64 0, i64 0), i32 %fi8) nounwind
34   %fi10 = call i32 @injectFault0(i32 10, i32 0, i32 %6)
35   br label %return
36
37 return:                                           ; preds = %bb2
38   call void @postInjections()
39   ret i32 undef
40 }
```

*Fig. 4:* LLVM IR of Figure 3 instrumented with fault injection calls

*TABLE I:* Difference between LLVM intermediate code and Assembly Language, and the impact on fault injection

| LLVM Instruction | Assembly Language Instruction | Mapping (if possible) |
|---|---|---|
| The GetElementPtr (GEP) instruction does address computation which is supplied to the load and store instructions for memory access. | A set of add and multiply instructions that computes the address | A fault in the GEP instruction translates to a fault in one of the add or multiply instructions |
| The PHINode instruction is inserted when the stack allocation and deallocation is promoted to registers, to choose between values merging from different basic blocks. | Stores to the stack in place of the PHINode instruction, and copy instructions in place of the source instructions in the corresponding basic blocks | A fault in the PHINode instruction translates to a fault in the store instructions, or the copy instructions corresponding to the sources of the PHINode instruction |
| Function call | PUSH/POP instructions for Caller/Callee saved registers before and after a function call, and Stack pointer stores return address | None since these instructions do not exist in the LLVM intermediate code. |
| Conditional branch instructions | Jump instructions where the Instruction Pointer register is fault injected | None since the branch instruction in LLFI is not fault injected, as it does not have a return value |

subsequent use in the program. We chose the following data type categories for splitting the results in order to study the correlation with data types.

1) Pointer data: Prior work has found that there is a high probability of SDCs when the fault affects the lower order bits of the pointer variable [17].
2) Control data: Prior work has found that a fault in control data may cause a control deviation, which might egregiously affect the computation [18], [19].

We separated the faults injected into those that affected data items that were present in the backward slice of these two types of data. The backward slice of a particular variable consists of all instructions that affect the output of the variable through a control or data dependency [20]. A fault occurring in the backward slice of a variable would be likely to propagate to the variable, and hence placing a detector at that variable would likely detect the fault. This leads to four classification categories as shown in Table II. Note that the faults are classified into these categories based on the dynamic execution, and hence each category is mutually exclusive.

**Example:** We now explain the correlation between an EDC and the data type, using a function `conv422to444` from the example code shown in Figure 5. This example is based on the MPEG benchmark, from the Mediabench Benchmark. The function `conv422to444` converts from YUV 4:2:2 subsampling (U and V components are sampled at half the rate of Y component) to YUV 4:4:4 (all components sampled at same rate). An EDC is caused by a fault in the loop termination condition `B2` and it belongs to the category of control data. However, faults in `B3` and `P1` cause Non-EDCs, which are also control and pointer data, respectively.

*TABLE II:* Classification of faults according to the backward slices of data categories (explained using example in Figure 5)

| Data Category | Explanation | Eg. |
|---|---|---|
| *Pointer and Control* | The fault (a) directly affects or propagates to a pointer, before affecting control data (deviation or backward slice) later in the execution OR (b) affects backward slice of control data (without causing a flip), and then propagates to pointer data | B1 |
| *Pointer and No Control* | The fault affects pointer data, but is not present in the backward slice of control data. | P1 |
| *Control and No Pointer* | The fault either causes a branch flip without/before affecting any pointer data, or affects control backward slice without affecting any pointer data. | B2 |
| *Neither Control nor Pointer* | The fault gets masked before being classified as belonging to the backward slice of control or pointer data. | - |

```
1  void conv422to444(char *src, char* dst, int width, int
       height, int offset) {
2      ...
3      if(dst < src + offset)   //B1
4        return;
5      for(j=0; j < height; j++)    //B2
6       for(i=0; i < width; i++) {
7            i2 = i<<1;
8            im1 = (i < 1) ? 0 : i-1; //B3
9            ...
10           dst[i2] = Clip[(21*src[im1])>>8]; //P1
11       }
12     ...
13 }
```

*Fig. 5:* Example Code of EDC versus Non-EDC data

## V. Implementation

In this section, we first explain the implementation of our fault injector. We then present the tracing mechanism to identify correlation between fault outcomes and data categories.

**LLFI:** We implemented the fault injector LLFI [2], and the tracing mechanism as custom passes in the LLVM compiler version 2.9. First, the application source code is compiled into LLVM IR along with the `mem2reg` optimization (i.e., promote loads/stores to registers). Second, the IR is statically instrumented with calls to our custom fault injection function, as explained in Section III-B. Third, the IR is statically instrumented with calls to the custom trace function, for control and pointer data (see below).

**PIN Fault Injector:** We build a fault injector at the assembly code level using PIN [9], a binary instrumentation framework for X86 processors. We use the PIN fault injector to quantitatively validate the accuracy of LLFI versus assembly level fault injection. The PIN fault injector is based on the same fault model used for LLFI. We inject a single bit flip into the write register of one dynamic instance of an assembly

[2]LLFI is available at http://github.com/DependableSystemsLab/LLFI

instruction chosen at random from the set of all instructions at runtime. The number of fault injections and the fidelity threshold value were kept the same between LLFI and the PIN fault injector. We present the results of this experiment in Section VI-B.

**Data Tracing:** After injecting the fault, we trace its propagation in the program as follows:

1) Instrumentation for trace collection: All relevant instructions were instrumented to record their result. In the case of control instructions, these include the branch target instructions [3] and the branch control variables. Branch control variables are those that make up the branch predicate. We trace the branch control variables to identify faults that affect the predicate but do not flip the branch. For pointer tracing, we instrument the instructions computing array indices. Further, we tag the pointers, to classify faults that occur at them as pointer deviation.
2) Trace Collection: During the fault-free run of the application, the result along with the `id` of the instructions instrumented above, is recorded. The same process is followed for each run of the fault injection experiment. We obtain three trace files for each fault injection run, namely the branch target instructions, the branch control variables, and the array index instructions.
3) Trace Comparison: At the end of each run of the fault injection experiment, we compare the faulty traces with the traces obtained during the fault-free execution, and classify the fault into one of the four categories mentioned in Table II. For example, a fault that differs in branch target trace, but does not differ in the pointer trace, is in category *Control and NoPointer*.

## VI. Results

In this section, we first present the benchmarks and the fidelity metric along with the threshold values. We then present the results of the quantitative comparison of LLFI with respect to the PIN fault injector, followed by the results of fault injection experiments, split according to the data categories identified in Table II.

### A. Benchmarks

We performed the fault injection for six applications of MediaBench I and II [13], [21]. This is a commonly used suite of soft-computing applications pertaining to multi-media processing. Three of the benchmarks, namely JPEG, MPEG2 and H264 decoders, use Peak-Signal-to-Noise Ratio (PSNR) between the faulty and fault-free decoded images as the fidelity metric [12]. JPEG is an image decoder, while MPEG2 and H264 are video decoders. The other three benchmarks are speech decoders - G721, GSM and ADPCM, and they use Segmental SNR as the fidelity metric [12]. We use PSNR of 30, and Segmental SNR of 80 as fidelity threshold values to differentiate EDCs from Non-EDCs [4].

[3]Instructions and the result of instructions are the same in SSA form, and hence we use them interchangeably

[4]We use Segmental SNR of 30 for ADPCM alone

We injected 1000 faults per benchmark using LLFI. We observed a 100% activation rate, i.e., all injected faults are activated. The EDC rates for all benchmarks are within an error bar of 2.2%, at the 90% confidence interval.

### B. Quantitive Validation of LLFI

Figure 6 quantifies the difference in fault outcomes between fault injection done using PIN versus LLFI. For all benchmarks, the EDC and Non-EDC rates are similar or higher for LLFI versus the PIN injector. This shows the validity of performing fault injections at the intermediate code level. On average, LLFI has an EDC rate of 6.4% versus 3.3% for the PIN fault injector, and a Non-EDC rate of 42.8% versus 23.9% for the PIN fault injector. A significant fraction of faults in the PIN experiment are benign or result in crashes. The average EDC rate is skewed towards the high EDC rate in ADPCM (which is around 23%). By excluding ADPCM, the average EDC rates is 3% for LLFI versus 2% for PIN. Unlike the other benchmarks, ADPCM has around 750 lines of code, with the main functionality of the benchmark concentrated in a single function. This function has many branch instructions, which leads to a high number of SDCs.

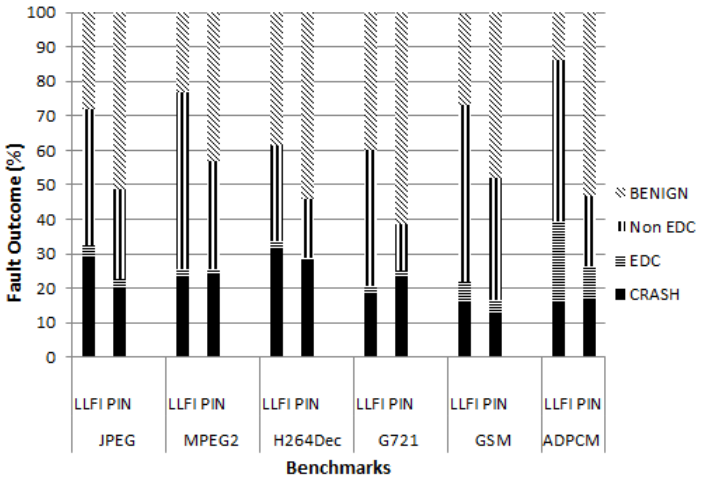


*Fig. 6: %* of Fault-Injection outcomes between LLFI and PIN fault injector, for all benchmarks

**PIN Injector:** To better understand the difference between the results of LLFI and PIN injector, we studied the correlation between faults at specific registers and the fault outcomes in the PIN injector. Faults in the instruction pointer (`IP`) and the stack pointer (`SP`) registers predominantly resulted in crashes. However, a small number of faults in the `IP` register resulted in EDCs (around 0.4% out of the total fault injections). These faults do not have a corresponding mapping in LLFI (difference 4 in Table I).

The high number of benign outcomes in PIN was due to faults affecting the `RFLAGS` register. This register is written in the case of the `test` instructions which is the predicate used for conditional branching. `RFLAGS` has 64 bits, each representing specific flags or reserved bits. The flag usually tested is the `zero` flag in the corresponding conditional branch

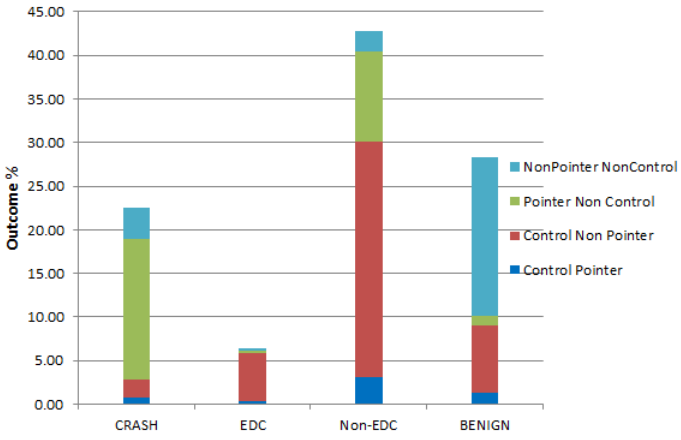


*Fig. 7:* Fault-Injection outcomes as per different data categories, across benchmarks under low fidelity threshold

instruction. Hence, a single bit flip which affects only one of these flags, does not cause an error unless the `zero` flag bit is flipped.

### C. Fault Injection Results

Figure 7 shows the average results across benchmarks for different failure types. From the figure, the average EDC rate across applications is 6.4%, while the Non-EDC rate is 42.8%. The crash rate is 22.5% and the benign fault rate is 28.3%. From the overall results, one can observe that EDCs constitute a small, but non-trivial fraction of the application outcomes. In fact, only 8% of the errors that do not crash the application result in EDCs. This shows that blindly detecting all non-crashing errors would result in significant wastage of energy and time, as many of them do not cause EDCs.

We observe the following trends about the correlation between the data items monitored (control and pointer) and the fault outcome from Figure 7.

**R1. Control Non Pointer**: An EDC outcome is highly correlated with a deviation in a data item belonging to the control backward slice (i.e., control non-pointer). These data items are usually loop termination conditions. Although Non-EDCs are also highly correlated with faults in control non-pointer, one difference between an EDC and a Non-EDC outcome is based on the amount of data affected by the branch deviation.

**R2. Pointer Non Control**: Data items that are in the backward slice of pointer data, but not that of control data (i.e., pointer non-control) are highly correlated with crashes. However, such faults can also result in SDCs (i.e., both EDCs and non-EDCs), especially if the fault affects the low-order bits of the pointer variable. A subset of the faults affecting low order bits cause EDCs.

**R3. Non Control Non Pointer**: Faults that do not belong to the backward slice of control or pointer data are usually benign. These faults usually get masked before being classified as belonging to these backward slices.

**R4. Control Pointer**: Although only a small fraction of faults cause deviations in the backward slices of both control and pointer data (around 5.8%), there is a high probability of such faults resulting in an EDC or SDC (e.g., around 62% of such faults result in an SDC outcome).

## VII. Conclusion

Soft computing applications can tolerate most errors that result in deviations in output or Silent Data Corruptions (SDCs). However, they do not tolerate outcomes that deviate significantly from the fault-free outcome, e.g. major glitches in decoded video. We classify such outcomes as Egregious Data Corruptions (EDCs). We performed a study to identify the correlation between faults in specific data types, and EDC outcomes, by building an LLVM based fault injector, LLFI.

Using LLFI, we perform fault injection experiments and monitor the categories of control and pointer data during the fault injection, to observe their correlation to the fault outcomes. We find that the class of faults affecting data belonging to the backward slice of control non-pointer cause the maximum number of EDCs. This data categorization helps us identify sensitive data in code, and hence identify detector placement locations.

**Acknowledgements**: This work was supported in part by a Discovery grant and an Engage Grant, from the National Science and Engineering Research Council (NSERC), Canada.

## References

[1] N. P. Carter, H. Naeimi, and D. S. Gardner, "Design techniques for cross-layer resilience," ser. DATE '10, 2010, pp. 1023–1028.
[2] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," ser. DATE '10, pp. 335–338.
[3] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: an architectural framework for software recovery of hardware faults," ser. ISCA '10. ACM.
[4] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: saving dram refresh-power through critical data partitioning," ser. ASPLOS '11. ACM, 2011, pp. 213–224.
[5] M. Carbin and M. C. Rinard, "Automatically identifying critical input regions and code in applications," ser. ISSTA '10. ACM, 2010, pp. 37–48.
[6] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "Ersa: error resilient system architecture for probabilistic applications," ser. DATE '10, pp. 1560–1565.
[7] L. Zadeh, "What is soft computing?" *Soft computing*, vol. 1, no. 1, pp. 1–1, 1997.
[8] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," ser. CGO '04.
[9] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.
[10] U. Schiffel, A. Schmitt, M. Susskraut, and C. Fetzer, "Slice your bug: Debugging error detection mechanisms using error injection slicing," in *EDCC '10*, pp. 13 –22.
[11] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults," ser. ASPLOS '12. ACM, 2012, pp. 123–134.
[12] X. Li and D. Yeung, "Application-level correctness and its impact on fault tolerance," ser. HPCA '07, feb., pp. 181 –192.
[13] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *MICRO 30*. IEEE Computer Society, 1997, pp. 330–335.
[14] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability,*, vol. 51, no. 1, pp. 63 –75, mar 2002.
[15] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.
[16] S. D. Toit. (2009, May) Why we chose llvm. [Online]. Available: http://software.intel.com/en-us/blogs/2009/05/27/why-we-chose-llvm/
[17] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," ser. DSN '12. IEEE Computer Society, pp. 181–188.
[18] D. Thaker, D. Franklin, J. Oliver, S. Biswas, D. Lockhart, T. Metodi, and F. Chong, "Characterization of error-tolerant applications when protecting control data," in *IEEE Intl. Symposium on Workload Characterization*, oct. 2006, pp. 142 –149.
[19] A. Sundaram, A. Aakel, D. Lockhart, D. Thaker, and D. Franklin, "Efficient fault tolerance in multi-media applications through selective instruction replication," ser. WREFT '08. ACM, 2008, pp. 339–346.
[20] M. Weiser, "Program slicing," ser. ICSE '81. IEEE Press, pp. 439–449.
[21] J. Fritts, F. Steiling, and J. Tucek, "Mediabench ii video: expediting the next generation of video systems research," *Embedded Processors for Multimedia and Communications II, Proceedings of the SPIE*, vol. 5683, pp. 79–93, 2005.