Error Detector Placement for Soft Computation

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

***Abstract*—The scaling of Silicon devices has exacerbated the unreliability of modern computer systems, and power constraints have necessitated the involvement of software in hardware error detection. At the same time, emerging workloads in the form of soft computing applications, (e.g., multimedia applications) can tolerate most hardware errors as long as the erroneous outputs do not deviate significantly from error-free outcomes. We term outcomes that deviate significantly from the error-free outcomes as Egregious Data Corruptions (EDCs).**

In this study, we propose a technique to place detectors for selectively detecting EDC causing errors in an application. We performed an initial study to formulate heuristics that identify EDC causing data. Based on these heuristics, we developed an algorithm that identifies program locations for placing high coverage detectors for EDCs using static analysis. We evaluate our technique on six benchmarks to measure the EDC coverage under given performance overhead bounds. Our technique achieves an average EDC coverage of 82%, under performance overheads of 10%, while detecting 10% of the Non-EDC and benign 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]. Soft computing applications have gained increasing prominence, and researchers have predicted that future workloads will belong primarily to this category [8].

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 that deviate significantly from the fidelity metric, i.e., unacceptable outcomes.

The error tolerance of soft computing applications does not mean that they are resilient to all errors. In particular, an error in a soft computing application may or may not lead to an EDC. If it will lead to an EDC, then the application needs to be stopped, as otherwise, its output will be unacceptable. On the other hand, if the error will not lead to an EDC, it is better to let the application continue rather than perform wasteful detection and recovery, and incur unnecessary overheads. This overhead will become more prominent as error rates increase, as they are predicted to do so in future processors.

Our goal is to efficiently place error detectors in soft-computing applications in order to detect errors early (thus avoiding egregious outcomes), at the same detecting only those errors that lead to EDCs (thus avoiding wasteful recovery). An error detector is an assertion or check on one or more data variables in the application. We develop heuristics that determine where to place error detectors for avoiding EDCs, using fault injection and static analysis of the application's code. While we use fault injection to develop the heuristics, we do not require fault injection to apply the developed heuristics to new applications. This is because our heuristics are based on static and dynamic properties of the application's code, and do not rely on semantic knowledge of the application. Note that fault-injection is a time intensive process for large applications, and hence it is desirable to avoid it (if possible).

Prior work [9], [10], [11] has investigated the problem of optimal error detector placement. However, these techniques focus on placing detectors to minimize the error detection latency or to detect specific failures such as safety violations. In particular, they do not consider optimizing the detector placement for minimizing the EDC rate, which is important for soft computing applications. As we show later in this paper, minimizing the EDC rate leads us to different placement decisions than if we had optimized for minimizing the number of Silent Data Corruptions (SDCs), which constitute any deviation from the correct output (not only egregious ones) [12].

We make the following contributions in this work:

1) We perform fault injection into soft computing applications, and distinguish EDCs from the set of SDCs. Based on the injections, we develop heuristics for identifying EDC-prone regions of code or data, which are appropriate candidates for detector placement.
2) We develop a systematic algorithm based on these heuristics, that (a) ranks the data according to their EDC causing nature, based on static analysis (b) uses a greedy approach that combines the static information, with the dynamic execution profile, to choose the appropriate set of EDC data or code for placing detectors. Our algorithm takes as input the application source code, the acceptable

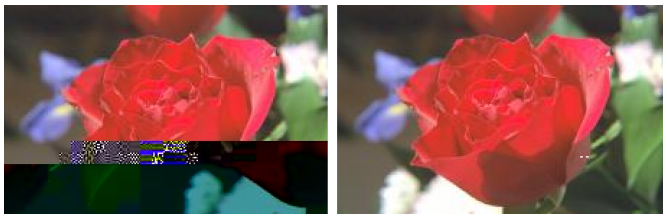

Fig. 1: The EDC causing fault decoded image (left) versus Non-EDC causing fault decoded image (right) from the JPEG decoder

performance overhead, and the execution profile data of the application, and identifies the locations to place detectors in the program, i.e., data or instructions.

3) We implement the algorithm within the LLVM compiler, and evaluate its accuracy through fault-injection. We find that the detectors placed by the algorithm provide EDC coverage of 82% under 10% performance overhead, while providing a Non-EDC and benign coverage of only 10%.

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 [13]. For example, the fidelity metric used by speech decoders is Segmental SNR. 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,i.e., SDCs that do not belong to the category of EDCs. For brevity, we use the terms EDCs and EDC causing errors interchangeably throughout this paper.

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 or are of very poor image quality. In other cases, where we cannot rely on human perception, we set the fidelity threshold to be such that around 30% of the most egregious SDCs are categorized as EDCs (see Section VI).

The example in figure 1 shows the faulty decoded images of the JPEG decoder (part of Mediabench [14]), when a fault is injected into the program. The fidelity threshold is Peak Signal to Noise Ratio (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.

Example: We now explain the correlation between an EDC and the program characteristics, using a function `conv422to444` from the example code shown in Figure 3 in Section IV. 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).

Challenge: Prior research has shown that faults in data constituting higher dynamic execution time are more likely to cause SDCs [12]. In other words, SDC causing code tends to be on the hot paths of the application. However, EDCs are caused by a large deviation in output, and are not necessarily caused by faults in data on the hot paths. For example, in Figure 3, the longest running statements are the lines 8 to 11, the ones within two nested `for` loops. A fault at the branch `i < 1` at `B4`, or at the pointer data `P1`, causes an SDC but not an EDC. However, a fault occurring at loop termination conditions `B2` and `B3` cause an EDC.

Therefore, to maximize the coverage for EDCs, detectors should be placed at code regions or data that have the highest impact on the application's output. The main challenge in detecting EDCs is coming up with a general algorithm to identify such code or data. Further, the algorithm should be based only on the static code of the program and its execution profile, and not require fault injections, which are expensive. This is the main challenge we address in this paper.

III. INITIAL STUDY

This section describes our initial study for identifying potential locations to place detectors for EDC causing faults through fault injection experiments. We present our fault model in section III-A and describe the fault injection experiment in section III-B. Finally, we describe the results of the fault-injection experiment in section III-C. Based on these results, we develop heuristics for finding the EDC causing data in the program, as explained in Section IV.

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. These factors get exacerbated as the supply voltage is reduced for saving power in processors.

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 [15]. As in prior work, we do not consider faults in floating point registers [12] - this is part of future work.

B. Fault Injection Experiment

For the fault-injection experiments, we use LLFI, a program level fault-injection tool we built for performing the fault injection experiments [16]. LLFI works at the LLVM

compiler's intermediate code level [17], and allows fault-injections to be performed at specific program points, and into specific data types. It also 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. In our prior work [16], we have quantitatively verified the accuracy of LLFI compared to binary code level fault injection, for soft-computing applications.

We performed the fault injection on six applications of MediaBench I and II [14], [18]. This is a commonly used suite of soft-computing applications pertaining to multi-media processing. Three of the benchmarks - JPEG, MPEG2 and H264 decoders, use PSNR as the fidelity metric. The other three benchmarks are speech decoders - G721, GSM and ADPCM and they use Segmental SNR as the fidelity metric. We use a PSNR value of 30, and Segmental SNR value of 80 as fidelity threshold values to differentiate EDCs from Non-EDCs.

In each run, a fault, i.e., a single bit flip, was injected into the destination register of exactly one dynamic instance of an instruction chosen at random (among all the executed instructions), and the outcome of the fault was classified by comparing the final output 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. We perform 1000 fault injections per benchmark and classify the outcomes into crash, benign, EDCs and Non-EDCs

After injecting the fault, we monitored the program at selected data items, in order to determine if the values of the data items exhibit a deviation from their fault-free values. In other words, we compare the value of the data items between the fault-free and fault-injected runs (we collect these values by instrumenting the program with the LLVM compiler). This information will help us determine whether to place detectors at the variable.

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 subsequent uses in the program. This also helps us formulate heuristics later in this paper.

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 [12].
2) Control data: Prior work has found that a fault in control data may cause a control deviation, which might egregiously affect the computation [19], [20].

We separated the faults injected into those that affected data items that were present in the backward slice of either 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 [21]. 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 I.

TABLE I: Classification of faults according to the backward slices of data categories (explained using example in Figure 3)

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	-

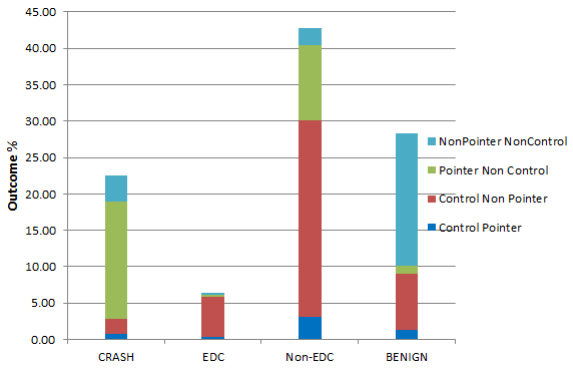

Fig. 2: Fault-Injection outcomes as per different data categories, across benchmarks under low fidelity threshold

C. Fault Injection Results

Figure 2 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 errors would result in significant wastage of energy and time, as many of them do not cause EDCs. This is the main reason why we develop targeted techniques for detecting EDCs in soft computing applications.

We observe the following trends about the relationship between the data items monitored and the fault outcome from Figure 2. We will use these observations for formulating heuristics in Section IV.

R1. Control Non Pointer Faults in data items belonging to the control backward slice (third category in table I) are highly likely to lead to EDCs. These data items are usually loop termination conditions, and are further discussed in Section IV.

R2. Pointer Non Control Data items that are in the backward slice of pointer data, but not control data (i.e., pointer non-control) are responsible for most 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.

R3. Non Control Non Pointer Faults in data items that do not belong to the backward slice of control or pointer data are mostly benign. Therefore, we do not consider this data category in formulating heuristics.

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 SDC (e.g., around 62%). We take this into account when formulating heuristics in Section IV.

IV. HEURISTICS

We formulated heuristics to identify detector placement points for EDCs, on the basis of the fault injection experiments in Section III. Our heuristics are generic, in that they can be applied to any soft-computing application, and *do not require semantic knowledge of the application.* Further, while we use fault injection to formulate the heuristics, the heuristics themselves do not need fault injection to be applied to new applications.

All of these heuristics have the common characteristic of being dependent on the size of the data being affected, either within the branches or in downstream computations. We unify these heuristics using a ranking expression in our algorithm explained in Section V.

We explain the heuristics with the code in Figure 3 as a running example. This code is based on the MPEG video decoding benchmark from the Mediabench benchmark suite. However, for elucidation purposes, we have added extra code to these functions (we explain what these are later). The `store_ppm_tga` function stores the decoded image in a `ppm` file. The `Show_Bits(N)` function returns the next `N` bits of the image, without advancing the pointer.

We divide the problem of formulating heuristics for identifying detector placement points into two steps. First, we identify functions in the program that are likely to result in EDCs when affected by faults. Second, we identify statements (and variables) within a function at which detectors need to be placed in order to detect EDCs.

A. Step 1: Function Identification

We first identify program functions in which we need to place detectors, based on whether the functions have side effects. A side-effect free function has the following two characteristics, both of which must be satisfied:

1) Statements within these functions do not modify global variables, files and pointers, though they may *read* them.
2) The functions have a return value and this is the only result of the function used by its caller function

We call such functions Optimized EDC Functions (OEF). For example in Figure 3, `Show_Bits()` is an OEF, as it satisfies the conditions outlined above. Once an OEF call is identified as EDC-causing, it suffices to place a detector at the return value of the particular call. No other detectors are required for the OEF, and hence the name Optimized EDC. We find that EDCs are caused by only *certain* calls to OEFs, and we formulate a heuristic for identifying such OEF calls.

```
1  void conv422to444(char *src, char* dst, int width, int
       height, int offset){
2      ...
3      w = width>>1;
4      if(dst < src + offset)   //B1
5        return;
6
7      for(j=0; j < height; j++) {   //B2
8       for(i=0; i < width; i++) {   //B3
9            i2 = i<<1;
10           im1 = (i < 1) ? 0 : i-1; //B4
11           ...
12           dst[i2] = Clip[(21*src[im1])>>8]; //P1
13      }
14      if(j + 1 < height) {   //B5
15           src += w;       //P2
16           dst += width;
17       }
18    }
19     ...
20 }
21 void store_ppm_tga(int width, int height){
22   int i, j, singlecode;
23   char *u444[NUMFRAMES];
24   int *code[NUMFRAMES], codeframes[NUMFRAMES];
25   ...
26   //int *bitlocn[NUMFRAMES] is global
27   for(i=0; i < NUMFRAMES; i++){ //B6
28     for(j=0; j < width; j++)
29       singlecode += Show_Bits(bitlocn[i][j]); //C0
30     codeframes[i] = singlecode;
31   }
32   for(i=0; i < NUMFRAMES; i++) //B7
33     for(j=0; j < width; j++)   //B8
34       code[i][j] = Show_Bits(bitlocn[i][j]); //C1
35   ...
36   singlecode = Show_Bits(8); //C2
37   ...
38   //char *source[] is global
39   if(CHROMA_FORMAT == YUV422){   //B9
40     for(i=0; i < NUMFRAMES; i++)   //B10
41       conv422to444(source[i], u444[i], width, height,
           offset); //C3
42   }
43   ...
44 }
45 main(){
46   ...
47   store_ppm_tga(width,height);
48   ...
49 }
50 unsigned int Show_Bits(int N){
51   //ld is a global struct
52   return ld->Bfr >> (32-N);
53 }
```

Fig. 3: Example Code for Function and Data Categorization

H1: The likelihood of an EDC due to a fault in an OEF increases as the amount of data affected by its return value increases.

We formulated this heuristic based on the results R1 and R2. By the definition of OEFs, the data modified within these functions is local to the function. Therefore, the data modified by an OEF call is dependent on the propagation of the function's return value. For example, `Show_Bits()`, which is an OEF, is called at three places in the code, namely `C0`, `C1` and `C2`.[1] When a fault occurs in the OEF, the return value

[1]This function is called only in the manner specified in `C2`, in MPEG. We add the remaining calls, to explain examples seen across other applications.

of the function call at C1 affects only one element of the 2D code array. This fault does not cause an EDC. On the other hand, the function call in C0 is a loop carried dependency, and the singlecode variable is assigned to the elements of array codeframes in the outer loop. The return value from the C0 call thus influences a larger amount of data than the return value from call C1. Therefore, we will place a detector on the return value in call C0, but not on the return value in call C1.

Based on the heuristic H1 above, we identify the calls to OEFs on whose return values we place detectors. Note that this heuristic only applies to OEFs called within loops. When the OEF is not called within a loop, we do not place any detector at the return value. This is because such faults usually cause an EDC when they propagate to branches, and would be caught by detectors at those branches. For example, neither the call C2, nor its caller function store_ppm_tga is called within a loop. Hence, we do not place a detector after call C2.

The remaining functions are side-effect causing functions. For example, these are conv422to444 and store_ppm_tga, since they do not satisfy the conditions required for being classified as an OEF. We elaborate the heuristics applicable to such functions in the following section.

B. Step 2: Data Categorization

Within functions that are not OEFs, we found that faults affecting certain control and pointer data are highly likely to cause EDCs (based on results R1, R2 and R4).

Control Data: We formulated the heuristics for control data by analyzing the results R1 and R4. Control data can be divided into loop or function terminating branch conditions, and other branches, i.e., those that do not terminate loops or functions. For example, B1 is a function terminating branch, while j < height at B2 is a loop terminating branch. The heuristics are based on faults that either directly affect or propagate to these branches, and cause the branch to flip.[2]

H2. The EDC causing nature of the loop terminating conditions decreases, as we go deeper within nested loops. We formulated this heuristic based on the result R1. This is because the amount of data modified by outer loops is much larger than the data modified by inner loops. For example, a fault at branch condition i < NUMFRAMES at B7 has a higher likelihood of causing an EDC than one at branch condition j < width at B8, as it affects more elements of the array code.

H3. The likelihood of an EDC due to faults at function terminating conditions decreases as the inter-procedural loop nesting level increases, and as the amount of data affected in downstream computations within that function decreases.

We formulated this heuristic based on the result R1. The EDC causing nature of function terminating branches decreases, as the inter-procedural loop level increases. For example, the function terminating branch B1 has a loop level of 1, since the function conv422to444 is called within a loop at C3. Also, a fault at B1, causing a branch flip to true, abnormally terminates function execution, thereby missing the loop computations at B2. These two factors, i.e., the downstream loop computations and the low loop nesting level, contribute to a high likelihood of an EDC, when the fault occurs or propagates to B1. In cases where loops do not follow the function terminating conditions, we see that the faults do not cause EDCs.

H4. Branch conditions that do not terminate functions or loops are likely to cause EDCs if and only if the amount of data affected within the body of the branch is large.

We formulated this heuristic based on results R1 and R4:

1) When the branch body consists of assignments to pointers, or several elements of an array or aggregate structure, a fault occurring at the branch results in an EDC. In the above example, we place a detector after branch B5 since the body of the branch changes the pointers src and dst.
2) When the branch body consists only of a change to a single element of an array, or some local variable, a fault in the branch results in an SDC, but not an EDC. For example, the ternary condition i < 1 at B4 is a Non-EDC causing branch, since it only changes the index of one element in the array src, thereby corrupting the value of one element of array dst.
3) When a branch condition (that does not terminate loops or functions) has loops within it, a fault at the branch condition has a high likelihood of causing an EDC. This is because the amount of data modified is large in the loop body. For example, a fault causing a branch flip at B9 has a high likelihood of causing an EDC, since it causes the loop at B10 to be skipped, thereby affecting the computation of the entire array u444.

Pointer Data: Examples are pointer dereferences, accesses to specific elements within aggregate structures, and pointer assignments or arithmetic. Result R2 shows that the number of faults leading to crashes, SDCs and EDCs are high for pointer data that do not cause any control deviation. This pointer data usually occurs within loop bodies. As prior work finds [12], crashes are caused when a bit flip occurs in the high order bits of the memory access, whereas SDCs are caused when the bit flip is in the low order bits. However, we find that some pointer address computations are more likely to cause an EDC, and we formulate a heuristic to identify these computations.

H5. Faults in the low order bits of pointers pointing to larger sized data have higher likelihood of causing an EDC. We formulated this heuristic based on result R2. For example, faults in the low order bits of pointer data for src at P2 causes an EDC. However, a fault at the lower bits of the Clip, src or dst array indices at P1 causes an SDC, but not an EDC.

V. Approach

In this section, we first present the usage model for our technique, and then discuss our algorithms to identify program locations for high coverage detectors for EDC causing errors. These are based on the heuristics we developed in Section IV.

Usage Model: The goal of our technique is to preemptively detect EDC causing faults in soft computing applications, under a given performance overhead that the user is

[2]In our initial study, we found that faults causing branch flips are much more predominant than those that do not cause a flip for control data.

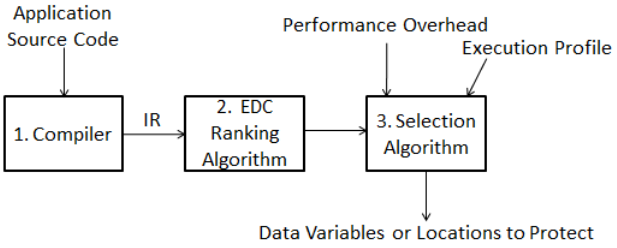

Fig. 4: Technique Workflow with required inputs

TABLE II: Attribute Values and Static EDC Rank for data items from example in Figure 3. Static EDC rank is calculated using equation 1 with the values of $\alpha = 4$, $\beta = 3$, $\mu = 2$, and $\gamma = 1$. Higher EDC rank implies higher likelihood of EDC

Data Item	OuterLoop Level	InnerLoop Level	DomLoop Level	DataWithin	EDC Rank
B1	0	0	2	0	6
B2	1	1	0	0	2
B4	3	0	0	0	0.1667
B5	2	0	0	2	0.5
P2	2	0	0	1	0.25
B6	1	1	2	0	5
C0	3	0	0	1/2	0.1667
C1	3	0	0	1/3	0.1667
B9	0	1	0	0	4

willing to tolerate. The technique requires as inputs from the user: (a) the application source code, (b) the maximum permissible performance overhead, and (c) the application's execution profile, under representative inputs.

The workflow of our technique is outlined in Figure 4, and consists of three steps. First, we compile the application source code using a standard compiler into an Intermediate Representation (IR). The IR should retain type information from the source code, and should be in Static Single Assignment (SSA) form [22]. 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. Second, we rank the application's data according to their likelihood of causing an EDC using an *EDC ranking* algorithm. Third, we choose the optimal data set for detector placement under the given performance overhead bound, using a selection algorithm that combines the obtained EDC ranks and the runtime profiling information. We describe the second and third phases of Figure 4 in Sections V-A and V-B.

A. EDC Ranking Algorithm

In this phase (step 2 in Figure 4), we first identify the initial dataset, i.e., the list of potential EDC causing data items based on the heuristics in Section IV. We then extract certain common attributes of these data items using static analysis. Finally, we formulate a ranking expression using these attributes, and rank these data items using our ranking expression.

Initial DataSet: The initial dataset consists of all the data categories identified through heuristics `H1` to `H5`. These are OEF calls, control data and pointer data. Note that this dataset contains EDC as well as Non-EDC causing data items. Using the heuristics, we formulated a ranking metric to rank these data items based on their tendency to cause EDCs (when faulty).

EDC Rank Characteristics: We discuss the characteristics of the EDC rank, and the rationale behind it. In Section IV, we found that data items that affect a larger amount of data have a higher likelihood of causing EDCs. *Hence, the EDC rank should be higher for data items affecting larger-sized data.* In other words, for any given data item `d`, the branch `b` it is control-dependent on, has a higher EDC rank than `d`. For example in Figure 3, the EDC rank should be such that $B2_{edcrank} > B5_{edcrank} > P2_{edcrank}$. We ensure these characteristics are satisfied through the computation of the attributes in the EDC rank equation, as explained below.

Attribute Extraction: The EDC rank of a data item depends on various attributes, which are extracted through static analysis of the program. The attributes and their values for the example in Figure 3 are shown in Table II. We explain the attributes below, using the example.

1) *OuterLoop Level* - The maximum level of loop nesting this particular data item is nested at. We extract this attribute based on heuristic `H2`. The outermost loop is at level 1, the next loop at level 2, and so on. For data items that are not loop or function terminating, the loop level is one level more than the number of loops it is nested within. This is to satisfy the EDC rank characteristic, and unify the attribute extraction across all data items. For example, branch `B4` in Table II has outerloop level of 3.
2) *InnerLoop Level* - The maximum level of loop nesting within this data item. We extract this attribute based on heuristics `H2` and `H4`. For example, branches `B2`, `B6` and `B9` have an innerloop level of 1.
3) *DomLoop Level* - The maximum level of loop nesting *dominated by* this particular data item, but excluding the innerloop level. The data item *d* dominates a loop if every path in the control flow graph from the start node to the loop should pass *d*. We extract this attribute based on heuristic `H3`. For example, the value for the function terminating condition `B1` is shown in Table II.
4) *DataWithin* - The amount of data affected by the data item. This applies to OEF calls, branches that are not loop or function terminating,[3] and pointer data. We extract this attribute based on heuristics `H1`, `H4` and `H5`. For pointer assignments and arithmetic, the numerical value of datasize is equal to the level of pointer indirection. In case of array accesses, the datasize is computed as 1/(1 + number of array indices). For example, the datasize for both pointer data `P2`, and OEF calls `C0` and `C1` is shown in Table II. Note that fractional values for the datasize might be masked in some cases due to the minimum value being 1 in the numerator. However, we find that few such cases affect the final value of the EDC rank in practice.

Note that the first three attributes refer to the *maximum nested level* of the loops, and not the number of loops. For scalability reasons, we only consider the intra-procedural loop nesting level when computing the outerloop level attribute for the EDC data set. This does not affect our results for most benchmarks.

[3]The attributes InnerLoop and DomLoop Level, already estimates the data affected by terminating conditions.

Static EDC Rank Expression: The EDC rank of a data item is the likelihood of an EDC outcome, given that a fault occurs at the data item or propagates to it. We formulate the rank expression using the attributes identified before:

$$\frac{max(\alpha * InnerLoop + \beta * DomLoop + \gamma * DataWithin, 1)}{max(\mu * OuterLoop, 1)} \quad (1)$$

where α, β, γ and μ are parameters quantifying the importance of the respective attributes, i.e., InnerLoop level, DomLoop level, DataWithin and OuterLoop level. To avoid zero values in the numerator and denominator, we assign the minimum value to be 1 in both parts.

We followed an educated trial and error method to assign the values for α, β, γ and μ. The values assigned are α = 4, β = 3, γ = 1 and μ = 2.

We explain the assignment of these values here. Recall that the EDC rank is higher for data items affecting larger sized data. The impact on the amount of data affected, is much higher for the first three attributes (*OuterLoop*, *InnerLoop* and *DomLoop*) than for the *DataWithin* attribute. This is because much higher amount of data is affected within loops than outside loops. Hence, we assign the lowest value for γ, which is set to 1. Further, as we go deeper down nested loops, the likelihood of an EDC outcome due to a fault in the data item decreases. Hence, we assign the highest value for α, the coefficient of the *InnerLoop* attribute, which is 4. The *DomLoop* attribute has the next highest value followed by *outerloop*. So, we assign the values β = 3, and μ = 2. We have experimented with other assignments in relation to each other, and find that the above assignment is the optimal one.

B. Selection Algorithm

In this phase, i.e., step 3 in Figure 4, we identify the optimal set of locations to place detectors in the program based on the EDC rank (from the previous phase), the allowed performance overhead and execution profile of the application. We use the profile data to maintain the bound on the performance overhead specified by the user, while accounting for the likelihood of a fault affecting the data item. We obtain the profile data by running the application with representative inputs provided by the user (see Section VI).

We model the problem of selecting the EDC data items as the 0-1 knapsack problem [23]. Each EDC data item d has an associated weighted EDC rank d_{wrank} (the objective function we maximize) and a performance overhead d_{po}, measured as the number of extra instructions that would need to be executed if the element is selected. Our goal is to select the items to put into the knapsack to maximize the rank subject to a given performance overhead. The weighted EDC rank is calculated using the following equation:

$$d_{\text{wrank}} = (norm(d_{\text{edcrank}}) + 1)/F_{\text{funcrank}} \quad (2)$$

where *F* is the function containing the data item *d*. The normalization function *norm*, converts the *edcrank* (obtained from previous phase) to a value between 0 and 1. The *funcrank* is the rank of the function in descending order of their execution time. We choose the set of detector locations (the knapsack), using the following criteria

$$maximize(\Sigma d_{\text{wrank}}) \text{ such that } \Sigma(d_{\text{po}}) \leq P \quad (3)$$

where *P* is the user specified maximum performance overhead.

A naive approach to solving the knapsack problem is a greedy one of choosing the item with the maximum weighted rank that satisfies the performance overhead constraint. However, a naive greedy algorithm may make a sub-optimal decision in choosing data items as it does not have a lookahead capability. We use a variant of the greedy algorithm that has a parameter controlling the function rank, and a lookahead window to avoid making a short-sighted, sub-optimal decision.

We explain the algorithm using an example. Let us consider five functions A, B, C, D and E, whose execution times are 10, 8, 6, 4 and 1 milli-seconds, respectively [4]. If we used a naive greedy algorithm, then the *funcrank* would be simply incremented as function execution times decreased. In this case, A, B, C, D and E would have respective *funcranks* of 1, 2, 3, 4 and 5. The selection algorithm would start filling the knapsack with data items of A in descending order of *edcrank*, followed by that of B, and so on, until the maximum performance overhead *P* is reached. Hence, the Non-EDC data in function A will get included, and we may miss the EDC data in the remaining functions, leading to a sub-optimal solution.

To overcome this problem, we use a *funcrange* parameter to increment the function rank. All functions having execution times within the *funcrange* have the same function rank. We also use a lookahead window with functions having the next higher function rank. Assuming *funcrange* with value 2, then functions A, B and C have a function rank of 1, D has a rank of 2, and E has a rank of 3. We explain how these ranks are obtained in algorithm in Figure 5. The selection window has functions A, B and C, while the lookahead window contains function D. Now, the knapsack is filled in descending order of d_{wrank} (where *d* is data items of *A, B, C and D*) until all the data items in the selection window are added. Next, the selection window slides ahead to D, and the lookahead window slides to E. The same process of filling the knapsack and sliding the window is repeated, until *P* is reached. As the *funcrange* parameter increases, more functions would have the same function rank. Hence, the choice of detector locations would be based on a larger set of data, and hence be more optimal than a naive greedy algorithm.

The algorithm to calculate the weighted EDC rank using *funcrange* of *N* is presented in Figure 5. It considers the functions in the program in decreasing order of their execution times. All functions within the *funcrange* have the same function rank. When a function whose execution time is outside the parameter is encountered, the function rank is incremented. If a function is an OEF, it is skipped (see Section V-A). After calculating the weighted EDC ranks for all the data items, the final set of EDC detector locations is computed using equation 3.

VI. Experimental Setup

In this section, we present the implementation details of our technique, followed by the benchmarks, the fidelity thresholds and the evaluation metrics.

[4] In actual implementation, we consider the number of dynamic instructions executed by the function as its execution time.

```
float funcrank = 1;
int funcrange = N;
map funcrankmap;
map EDCrankmap;
int main(){
  map weightedrankmap;
  function topFunc = Function with max exec time;
  for each function 'F' ranked in decreasing order of
       execution times{
    if(F is an OEF)
      continue;
    if(topFunc.exectime/F.exectime > funcrange){
      funcrank++;
      topFunc = F;
    }
    funcrankmap[F] = funcrank;
  }

  for each dataitem 'd' in initial DataSet{
    float weightedrank = calculateweightedrank(d);
    weightedrankmap[d] = weightedrank;
  }
}

float calculateweightedrank(dataitem d){
  Function F = d.getFunction();
  return ( (norm(EDCrankmap[d])+1)/funcrankmap[F] );
}
```

Fig. 5: Pseudo-code to show the calculation of weighted rank using *funcrange* = 'N' where *EDCrankmap* is map of static EDC ranks for all data items using equation 1

Implementation: We implemented the EDC ranking and the selection algorithm (steps 2 and 3 in Figure 4) as custom passes in the LLVM compiler version 2.9. First, the application source code is compiled into LLVM Intermediate Representation (IR) along with the `mem2reg` optimization (i.e., promote loads/stores to registers). Second, the IR is statically (a) analyzed to compute the static EDC rank for the EDC dataset, and (b) instrumented to place detectors identified using profile data[5] under the given performance overhead bound. Third, the instrumented IR is compiled into machine code using the LLVM compiler.

For the profile data, we need the user to provide representative inputs. However, the inputs are only used for calculating the performance overhead and the function rank. We have verified that the variation in EDC coverage is minimal across the provided inputs for the benchmarks we studied. We used *funcrange* of 5 in our experiments based on coverage results obtained by varying its value (see Section VII-C). The time required for our custom passes, is on average less than three seconds across the benchmarks.

The error detectors are derived by replicating the static inter-procedural backward slice of the EDC data item, and placing a comparison statement after the copy of the item. We do not consider reaching stores (for loads), and function pointers when computing the backward slice. Instead, we simulate these detectors by instrumenting the IR with trace calls at the locations chosen for detector placement. Hence, the coverage may be lower with actual detectors based on the backward slice. These trace calls record the values of the EDC data in a file, and a fault is detected if the fault-free and faulty trace files differ. The fault-free trace file is obtained by running the instrumented program on the same input, with no faults injected.

We measure the performance overhead of our detectors as the dynamic execution overhead of the extra code added (replicated code and comparison statements). We assume that faults do not affect detectors, and hence we do not inject faults into them. This is because we assume that only one fault occurs in each run of the application and a fault in the detector does not affect EDC coverage, as the worst outcome of such a fault is that it stops the program, and does not cause an EDC.

TABLE III: Characteristics of Benchmark Programs. Higher distortion (scaled difference) is more egregious, lower PSNR is more egregious.

Benchmark (Lines of C/C++ Code)	**Description**	**Input**	**Fidelity Metric (threshold value)**
BlackScholes (1661)	Compute option pricing using Black-Scholes Partial Differential Equation	Sim-large	Scaled difference of option prices (0.3)
X264 (37454)	Media Application performing H.264 encoding of video	test	Mean distortion of PSNR (as measured by H.264 reference decoder) and the encoded video's bitrate (0.017)
Canneal (4506)	Simulated cache-aware annealing to optimize routing cost of a chip design	Sim-dev	Scaled difference of routing cost between faulty and original version (0.026)
Swaptions (1428)	Price portfolio of swaptions using Monte Carlo Simulations	Sim-small	Scaled difference of swaption prices (0.00001)
JPEG (30579)	Image Decoder	testimg.jpg	PSNR between faulty and fault-free decoded images(30)
MPEG2 (9832)	Video Decoder	mei16v2.m2v	PSNR between faulty and fault-free decoded image set (30)

Benchmarks: We use four applications from Parsec [24], and two from Mediabench [14] for evaluating our technique. These are a mix of financial, multimedia and VLSI CAD applications, and have been used as soft computing applications in prior work [13], [25], [20], [26]. The benchmark characteristics are explained in Table III.

The majority of the programs are different from what we chose in our initial study, in which we only use Mediabench. We use only two programs from Mediabench (MPEG and JPEG) out of the six from our initial study. We do not use G721 and GSM, because their fidelity metric values show very slight variation, making it difficult to separate the EDCs from Non-EDCs, even manually. ADPCM is a small benchmark program with 740 lines of code, while H264Dec overlaps significantly with the Parsec benchmark X264, and hence both are skipped.

The other four programs are from the Parsec suite. Canneal is from Parsec 3.0, while the remaining three are from Parsec 2.1. We made small changes to some of the benchmark programs as follows: (1) For Blackscholes, we removed a loop that artifically increased the execution time and served no other purpose (also found by prior work [26]), (2) For Canneal, we applied a patch to reduce the load times on the serial version [27], (3) For Swaptions, we modified the code to include the first 240 trials of the monte-carlo simulation.

[5]We wrote a custom pass for obtaining profile data and for measuring the performance overhead, using LLVM basic block profiling pass.

We also removed specific assertion checks on function return values, as we do not need to 'check these checkers'. We did not find such code in any of the other benchmarks.

Fidelity Metrics and Threshold Values: We use the QoS metrics from prior work [26] as the fidelity metrics for the Parsec benchmarks. We distinguish EDCs from Non-EDCs using the fidelity threshold value (mentioned in parantheses in column 4 of Table III). This threshold value does not change between inputs. The distortion or scaled difference is the difference in absolute values between faulty and original fault-free value divided by the original fault-free value. For the Parsec benchmarks, we chose the fidelity threshold value such that 30% of the most egregious deviations from the SDC set are classified as EDCs. For MPEG and JPEG, we performed manual inspection of all the faulty outputs, and we noticed that EDCs were caused when the PSNR value was below 30, i.e., the images were severely distorted. Hence, we choose the value 30 as the fidelity threshold for these two programs.

Coverage Evaluation: We evaluate our technique by performing fault injection on the benchmark programs in Table III. We use our LLVM compiler based fault injector LLFI (used in the initial study in Section III-B) to perform the injections, and classify the outcomes as Crash, Benign, EDC and Non-EDC as explained in Section III-B. The applications are run using the LLVM Just-In-Time (JIT) compiler with the default optimization level of `O2`. We injected 2000 faults per benchmark. The EDC rates are statistically significant within an error bar of 1.32% at the 95% confidence interval.

We inject only one fault in each run, as we assume that transient faults are relatively rare events compared to the total execution time of an application. All injected faults are executed i.e., the instruction into which the fault was injected is executed by the program.

We measure the coverage under varying bounds on performance overheads, i.e., 10%, 20% and 25% (provided by the user) [6]. The EDC coverage is the fraction of detected EDCs out of total EDCs, while the Non-EDC and benign coverage is the fraction of detected Non-EDCs and Benign faults out of the total Non-EDC and Benign faults. We do not consider crashes as they are easily detected by program termination.

VII. RESULTS

In this section, we present the error outcome rates for the six benchmarks, followed by the coverage for EDC, and Non-EDC and Benign faults under varying performance overheads. We then study the effect of varying the *funcrange* parameter on the EDC coverage, and present the EDC coverage under varying fidelity threshold values. Finally, we present a quantitative comparison between our technique and a technique proposed in prior work.

A. Error Outcome Rates

Table IV shows the Crash, Benign, EDC, and Non-EDC rates for the fault injection experiments across the six programs. The average EDC rate across these applications is 4.03%, while the average Non-EDC and Benign fault rate is 57.57%. Although, this may seem to suggest that EDCs are not very important, one should keep in mind that these constitute the worst outcomes of the application. Further, the average Non-EDC rate is 21%, which is five times as much as the EDC rate. Hence, existing techniques that detect SDCs with high coverage will not be efficient for soft computing applications, because these techniques would also detect Non-EDCs with high coverage resulting in wasteful detection and recovery (we compare our technique with one such technique in Section VII-E).

TABLE IV: Percentage of Error outcomes in each benchmark

Benchmark	Crash (%)	Benign(%)	EDC (%)	Non-EDC(%)
BlackScholes	51.52	13.25	10	25.23
X264	28.4	64.9	2.72	4.53
Canneal	53.25	37.87	2.9	5.98
Swaptions	42.05	48.46	2.57	6.92
JPEG	29.27	30.38	4.03	36.27
MPEG2	25.85	22.83	2.01	49.37
Average	38.39	36.19	4.03	21.38

B. Coverage Under Varying Performance Overheads

EDC Coverage: Figure 6 shows the absolute EDC coverage across programs for different overheads. The average EDC coverage across the benchmarks is 82% at 10% overhead, 85% at 20% overhead, and 86% at 25% performance overhead. All applications except for Swaptions, have an EDC coverage of 80-100% (average being 96%) at 25% overhead. *Hence, our technique detects EDCs with high coverage (above 80%) in five out of six applications, with low overheads (10%).*

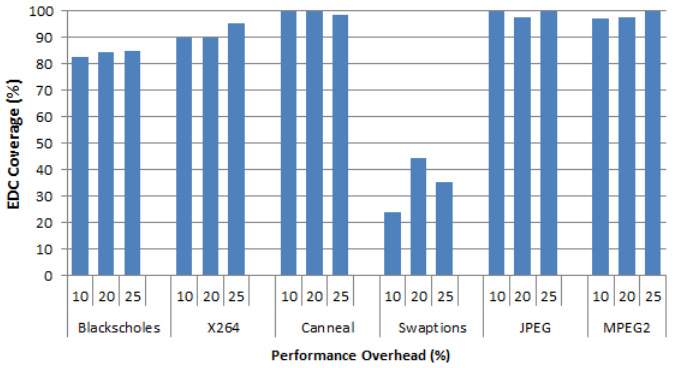

Fig. 6: EDC Coverage using our technique under performance overheads of 10%, 20% and 25%. Higher is better.

The lowest EDC coverage of our technique is for the Swaptions program (45%). It is interesting to note that Swaptions has a relatively low EDC rate of 2.5%. On further investigation, we found that many EDCs are caused by faults in the uniform random number generator function `RanUnif()`. The values returned by `RanUnif` are used in the rest of Swaptions as an input for Monte-Carlo simulations. However, this location is not chosen by our detector placement algorithm under the given performance overhead bounds. Our technique protects this function call only at 35% performance overhead bound, at which point the coverage increases to 80%. We believe this is an anomalous case as we do not see this behaviour in any of the other five benchmarks.

[6]We also measured coverage under 15% performance overhead, but do not present the results as they follow the trend of increasing coverages with higher performance overheads

Non-EDC and Benign Coverage: Figure 7 shows the Non-EDC and Benign coverage using our technique. Lower coverage is better as benign and Non-EDC faults are tolerated by the user, and we perform wasteful recomputation if these faults are detected as EDCs and recovered from. The average coverage is 10%, 16% and 17.6% under respective performance overheads of 10%, 20% and 25%. Further, the average benign fault coverage is lower than the Non-EDC coverage.

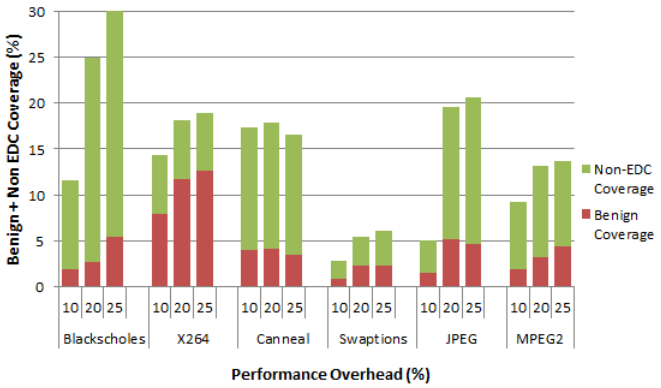

Fig. 7: Non-EDC and Benign Coverage for our technique, under performance overheads of 10%, 20% and 25%. Lower is better.

Summary: From Figures 6 and 7, one can observe that under 10% performance overhead, the average EDC coverage is 82%, while the Non-EDC and Benign coverage is about 10%. When the performance overhead is increased to 25%, the average EDC coverage is 86%, while the Non-EDC and Benign coverage is 18%. Using the absolute rates in Table IV, and considering the overall EDCs in the applications, this translates to correctly detecting 3.56% from the 4.05% EDCs, while wastefully detecting 10% from the 58% of Non-EDC and benign faults. If we consider the coverage to include all errors except EDCs, this corresponds to an increase in overall coverage from 95.95% without our detectors to 99.5% with our detectors, with 25% performance overhead across the six applications. *Therefore, our technique detects EDCs with high coverage, while detecting Non-EDCs and benign faults with low coverage, thereby efficiently differentiating EDC causing faults from the set of all faults in the application.*

C. EDC Coverage under varying funcrange values

The *funcrange* is a tunable parameter in the selection algorithm explained in Section V-B. Figure 8 shows the effect of *funcrange* on EDC coverage for the MPEG2 benchmark, for different performance overheads. With *funcrange* as 1 (priority to functions with higher execution time), the EDC coverage is 60%, 40%, and 50% under respective performance overheads of 10%, 20% and 25%. This is because of the short-sighted nature of the selection algorithm under small window sizes. As the *funcrange* increases to 5, the EDC coverage increases to almost 100%. However, there is almost no benefit in going to values of 7 and 9 [7]. We have observed a similar effect for other programs (not presented due to space constraints), and this is why we use *funcrange* as 5 in our experiments.

[7]There is a small dip in coverage from window size 5 to 7. We believe this is a statistical anomaly.

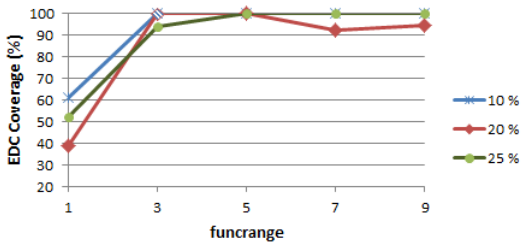

Fig. 8: Effect of varying *funcrange* on EDC Coverage, in MPEG2 under performance overheads of 10%, 20% and 25%

D. EDC coverage under varying fidelity threshold

As mentioned before, for the four Parsec benchmarks, we define EDCs to constitute 30% of the most egregious SDCs. In this section, we consider how the coverage varies if we consider *X%* of the most egregious SDCs to be EDCs, where *X* varies from 30 to 60. We do not consider JPEG and MPEG2, as they use absolute PSNR values for classifying EDCs.

Figure 9 shows the average EDC coverage for the four Parsec benchmarks as the EDC rate increases. As the % of SDCs classified as EDCs increases from 30% to 60%, i.e., the user or application has stricter constraints, the drop in coverage is at most 5% under the given performance overheads. This shows that our algorithm is reasonably robust to changes in fidelity threshold for classifying EDCs. We do not consider threshold values beyond 60% as at such values, EDCs are practically indistinguishable from SDCs (for our benchmarks).

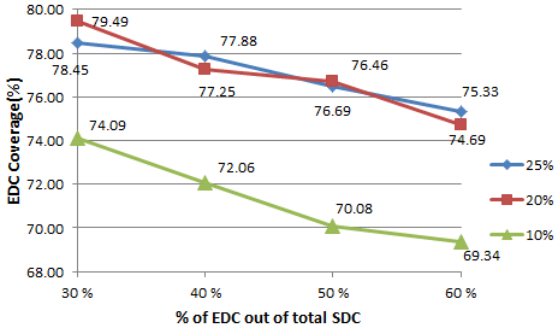

Fig. 9: Average EDC Coverage for the four Parsec benchmarks such that X% of most egregious SDCs are categorized as EDCs (under performance overheads of 10%, 20% and 25%)

E. Quantitative comparison with Prior Work

In this section, we quantitatively compare our technique with that of Sundaram et al. [20] who protect an application from soft errors by selective replication. Similar to our technique, they focus on multimedia applications that are a subset of soft-computing applications. However, unlike our approach, they do not distinguish between data that cause large output deviations and those that do not, and hence they protect all pointer and control data. In other words, they do not distinguish between SDC-causing errors and EDC-causing errors.

We implement Sundaram et al.'s technique by considering all control and pointer data as potential EDC data without any ranking or OEF tagging, and use our selection algorithm (with

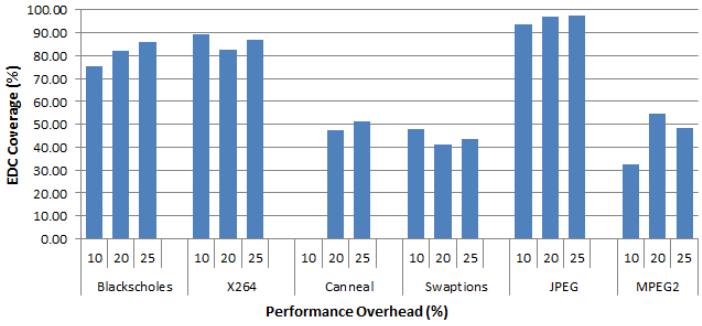

Fig. 10: EDC coverage by Selective Duplication Technique [20] under performance overheads of 10%, 20% and 25%

funcrange value of 5) to choose from the EDC data under the given performance bounds. The main difference with our earlier experiment is the absence of EDC data ranking that selectively detects EDCs from Non-EDCs and benign faults. Figure 10 shows the EDC coverage numbers under the given performance overhead bounds. The average EDC coverage is 56.4%, 67.5% and 68.9% under respective performance overheads of 10%, 20% and 25%, which is much lower than our technique (see Figure 6), for which the values are 82%, 85% and 86% respectively. The average Non-EDC and benign fault coverage varies from 11% to 17% under the given performance overhead bounds, which is comparable to our technique. *Thus, our technique has a higher EDC coverage than that of Sundaram et al. at nearly the same Non-EDC and benign fault coverage*.

VIII. Related Work

We classify related work into two areas, namely (1) identifying critical variables, and (2) placing error detectors in a program. Because we have already compared our technique with Sundaram et al [20] in Section VII-E, we do not consider this technique here.

Identifying Critical Variables A critical variable is defined as a variable that would cause a particular outcome (e.g., SDCs), when a fault occurs at that variable. There has been significant work on identifying critical variables for different kinds of failure outcomes in applications.

Cong and Gururaj [25] focus on identifying all critical variables in an application that can tolerate deviations in output (i.e., soft computing applications). Similar to our work, they also consider outcomes that cause large deviations from the correct output i.e., EDCs. They develop an algorithm to identify critical variables using static analysis, runtime profiling, and a runtime monitoring mechanism. Their approach differs from our work in two ways. First, they consider protecting critical variables, rather than placing detectors. As a result, they can incur much higher overheads than our technique, because protecting critical variables often involves duplicating the hot paths of the application. Second, their technique uses two versions of code - full duplication to ensure numerically accurate outputs, versus selective replication of only the critical variables. Based on the decision taken by the runtime monitoring mechanism, their solution switches to full duplication. Further, they do not present the EDC rates and the EDC coverage of the benchmark applications, which makes it difficult to quantitatively compare their technique with ours.

Identifying critical variables for software dependability has been explored from a software engineering perspective [28]. A critical variable, in this case, is based on its spatial and temporal impact, with respect to other software components. Similar to our work, their work also uses the static and dynamic properties of a program to quantify the impact of a variable. However, this technique also uses the failure rate of the variables in deciding if a variable is critical, which requires programmer knowledge and manual effort to calculate. In contrast, our technique is completely automated, and does not require any semantic information from the programmer.

Khudia et al. [29] use profile-based analysis along with symptom-detection to identify critical instructions for protecting against soft errors. They classify library and function calls as high-value instructions, and they tag as critical all instructions that produce the operands of the high value instructions i.e., those instructions in the backward slices of the high-value instructions. They perform memory and value profiling optimizations to reduce the overheads. However, they do not distinguish between EDC-causing and non-EDC causing errors, and hence their approach may perform wasteful detection and recovery for soft-computing applications.

Detector Placement Hari et al. [12] address the problem of detector placement (and detector derivation) for SDC-causing faults. The authors use a bottom up approach of analyzing the assembly code of specific programs to see what properties contribute to an SDC. Although we focus on EDCs, at a high-level, their work is similar to ours in terms of identifying program properties that cause a specific failure type. However, their work differs from ours in three ways. First, though they investigate detector placement locations, their main focus is on reducing the performance overhead incurred by instruction replication. Second, they rely on program specific functions in four out of six benchmarks to develop customized detectors (e.g., bit reversal and exponential functions). It is not clear how representative are these functions of general programs. Third, because of the use of high coverage customized detectors, their approach requires fault injection and manual extraction of specific program characteristics, at the machine code level, which is expensive.

Pattabiraman et al. [9] develop a set of heuristics for strategic placement of detectors to detect crash causing errors with low detection latency. The heuristics are calculated using the dynamic dependency graph (DDG) of the program, from one or more input sets. While these metrics help in placing detectors to preemptively detect crashes, their coverage for SDC (and EDC) causing errors is low. Further, because their approach requires constructing the DDG apriori, it has high performance overheads.

Hiller et al. [10], [11] focus on error detector placement from a software engineering perspective. They compare various techniques for identifying detectors, under different error models. However, they require fault injections to guide detector placement, which can be time consuming, and further do not focus on EDC-causing errors.

Snap [5] automatically identifies critical regions in code by grouping related input bytes into fields. It relies on application

code, and one or more inputs to see how targeted input fuzzing changes the behaviour of code. Code that causes large changes in output is classified as critical code, while code that induces small changes in output is classified as forgiving code. To some extent this is similar to our work on using program characteristics to identify detector placement points. However, their technique requires fuzzing, which is analogous to fault injection, and is hence time consuming.

Full duplication of programs using software redundancy will achieve close to 100% EDC coverage, at the cost of high performance overhead. However, there have been efforts to reduce the performance overhead using speculative redundant multithreading. An example is DAFT [30], in which the average performance overhead is reduced to 38%. DAFT [30] overcomes this by speculatively checking the results between the original and duplicate threads, and reducing the average performance overhead to 38% by asynchronously checking the results. However, due to the shared memory model, DAFT does not replicate loads and stores, and may miss faults in those instructions. Also, as the focus is on detecting SDCs with high coverage, and there is no effort to differentiate EDCs from this set. Therefore, DAFT can incur high Non-EDC coverage.

IX. Conclusion and Future Work

Soft computing applications 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).

In this paper, we first perform a fault injection study to identify the characteristics of EDC-causing errors in soft computing applications. Our initial study showed that these EDCs constitute only a small percentage of non-crashing errors, and hence it is crucial to selectively detect EDCs from the set of SDCs and benign faults. Based on the initial study, we develop heuristics for identifying EDC-prone regions of code and data in the application. We then develop a static analysis algorithm for identifying detector locations for detecting EDCs with high coverage, bounded by a given performance overhead. We find that the detectors placed by the algorithm achieve an average EDC coverage of 82% under 10% performance overhead, while detecting only 10% of the total Non-EDC and benign faults, for commonly used soft-computing applications.

As future work, we plan to implement the actual detectors and evaluate their coverage. We will also investigate optimizations to further lower the detectors' overheads, and to consider a wider range of applications.

Acknowledgment

We thank the anonymous reviewers of DSN for their comments that helped improve the paper. We also thank Sasa Misailovic for help with the Parsec x264 benchmark, and Jiesheng Wei for the helpful discussions. This work was supported in part by a Discovery grant and an Engage Grant, from the National Science and Engineering Research Council (NSERC), Canada. We thank the Institute of Computing, Information and Cognitive Systems (ICICS) at UBC for travel support.

References

[1] N. P. Carter, H. Naeimi, and D. S. Gardner, "Design techniques for cross-layer resilience," ser. DATE, 2010, pp. 1023–1028.
[2] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," ser. DATE, 2010, pp. 335–338.
[3] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: an architectural framework for software recovery of hardware faults," ser. ISCA, 2010, pp. 497–508.
[4] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: saving DRAM refresh-power through critical data partitioning," ser. ASPLOS, 2011, pp. 213–224.
[5] M. Carbin and M. C. Rinard, "Automatically identifying critical input regions and code in applications," ser. ISSTA, 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, 2010, pp. 1560–1565.
[7] L. Zadeh, "What is soft computing?" *Soft computing*, vol. 1, no. 1, pp. 1–1, 1997.
[8] P. Dubey, "Recognition, mining and synthesis moves computers to the era of tera," *Technology@ Intel Magazine*, pp. 1–10, 2005.
[9] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "Application-based metrics for strategic placement of detectors," ser. PRDC, dec. 2005.
[10] M. Hiller, A. Jhumka, and N. Suri, "On the placement of software mechanisms for detection of data errors," ser. DSN, 2002, pp. 135–144.
[11] M. Leeke, S. Arif, A. Jhumka, and S. Anand, "A methodology for the generation of efficient error detection mechanisms," ser. DSN, 2011, pp. 25–36.
[12] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," ser. DSN, 2012, pp. 181–188.
[13] X. Li and D. Yeung, "Application-level correctness and its impact on fault tolerance," ser. HPCA, 2007, pp. 181 –192.
[14] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," ser. MICRO, 1997, pp. 330–335.
[15] 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.
[16] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," ser. SELSE, 2013.
[17] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," ser. CGO, 2004, pp. 75–86.
[18] J. Fritts, F. Steiling, and J. Tucek, "Mediabench II video: expediting the next generation of video systems research," *SPIE - Embedded Processors for Multimedia and Communications II*, vol. 5683, pp. 79–93, 2005.
[19] D. Thaker, D. Franklin, J. Oliver, S. Biswas, D. Lockhart, T. Metodi, and F. Chong, "Characterization of error-tolerant applications when protecting control data," ser. IISWC, 2006, pp. 142 –149.
[20] A. Sundaram, A. Aakel, D. Lockhart, D. Thaker, and D. Franklin, "Efficient fault tolerance in multi-media applications through selective instruction replication," ser. WREFT, 2008, pp. 339–346.
[21] M. Weiser, "Program slicing," ser. ICSE, 1981, pp. 439–449.
[22] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *TOPLAS*, vol. 13, no. 4, pp. 451–490, 1991.
[23] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*, 2001.
[24] C. Bienia, S. Kumar, J. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," ser. PACT, 2008, pp. 72–81.
[25] J. Cong and K. Gururaj, "Assuring application-level correctness against soft errors," ser. ICCAD, 2011, pp. 150 –157.
[26] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," ser. ICSE, 2010, pp. 25–34.
[27] M. Roth. (2012, Jan.) Canneal speed up loading patch. [Online]. Available: https://lists.cs.princeton.edu/pipermail/parsec-users/2012-January/001270.html
[28] M. Leeke and A. Jhumka, "Towards understanding the importance of variables in dependable software," ser. EDCC, 2010.
[29] D. S. Khudia, G. Wright, and S. Mahlke, "Efficient soft error protection for commodity embedded microprocessors using profile information," ser. LCTES, 2012.
[30] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, "DAFT: decoupled acyclic fault tolerance," ser. PACT, 2010, pp. 87–98.