Error Detector Placement for Soft Computing Applications

ANNA THOMAS, University of British Columbia
KARTHIK PATTABIRAMAN, University of British Columbia

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. Our technique achieves an average EDC coverage of 82%, under performance overheads of 10%, while detecting 10% of the Non-EDC and benign faults. We also evaluate the error resilience of these applications under the fourteen compiler optimizations.

ACM Reference Format:
Anna Thomas and Karthik Pattabiraman, 2014. Error Detector Placement for Soft Computing Applications. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 25 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. 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 are challenging as power becomes a dominant concern in processor design [Carter et al. 2010].

Recently, there have been several proposals to selectively expose hardware faults to the software layer and tolerate them [Carbin and Rinard 2010; De Kruijf et al. 2010; Leem et al. 2010; Liu et al. 2011; Narayanan et al. 2010]. 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 [Zadeh 1997]. Soft computing applications have gained increasing prominence, and researchers have predicted that future workloads will belong primarily to this category [Dubey 2005].

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 pro-

Author's addresses: A. Thomas, (Current address) IBM , Toronto, Canada
K. Pattabiraman, Department of Electrical and Computer Engineering, UBC, Vancouver, Canada

DOI:http://dx.doi.org/10.1145/0000000.0000000

duced 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 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 [Hiller et al. 2002; Leeke et al. 2011; Pattabiraman et al. 2005] 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, 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) [Hari et al. 2012].

Recent work [Baek and Chilimbi 2010; Carbin et al. 2013; Samadi et al. 2013; Sampson et al. 2011] focus on approximate computing where accuracy of results is traded for performance benefits or lower energy consumption, by developing architectural or type-based solutions. For example, [Samadi et al. 2013] modifies the application code to generate approximate kernels to run on GPUs. Our work is also on similar applications which do not require precise results, but we study these applications from a fault tolerance and reliability perspective, through static analysis and execution profile.

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 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

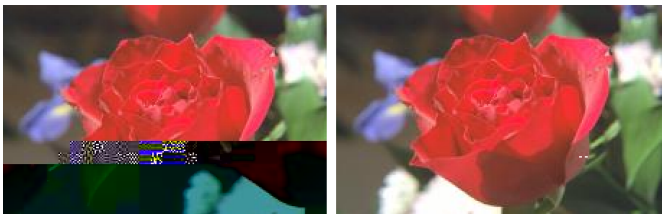

Fig. 1: The EDC causing fault decoded image (left) versus Non-EDC causing fault decoded image (right) from the JPEG decoder

EDC coverage of 82% under 10% performance overhead, while providing a Non-EDC and benign coverage of only 10%.

(4) We study the effect of individual compiler optimizations on the error resilience and vulnerability of soft computing applications, both with and without our technique. While compiler optimizations may enhance application performance, they might lower the application error resilience. We also identify safe optimizations, or those that do not affect the error resilience significantly, when detectors are placed in the application.

2. BACKGROUND

Egregious Data Corruption (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 5).

The example in figure 1 shows the faulty decoded images of the MediaBench JPEG decoder [Lee et al. 1997], 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.

Challenge: Prior research has shown that faults in data constituting higher dynamic execution time are more likely to cause SDCs [Hari et al. 2012]. 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, consider function `conv422to444` from the MPEG benchmark in Mediabench (Figure 2 in Section 3), which 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). 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.

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 focus on transient faults because they occur more frequently than permanent errors [Siewiorek 1991]. Further, rates of transient errors are projected to increase significantly due to the effects of technology scaling [Shivakumar et al. 2002]. Extending our fault model to permanent errors is a direction for future research.

Initial Study: To identify characteristics of EDC causing faults, we performed fault injection experiments on six applications of MediaBench I and II [Fritts et al. 2005; Lee et al. 1997]. These are video/image decoders - JPEG, MPEG2 and H264Dec, and speech decoders - G721, GSM and ADPCM. We use LLFI [Thomas and Pattabiraman 2013b], our program level fault injector that works at the LLVM compiler's intermediate code level [Lattner and Adve 2004], to perform the experiments.

Using LLFI, we injected faults (single bit flips) into pointers and control data, and monitored the fault propagation. The outcome of the fault was classified into Crash, EDC, Non-EDC and Benign, 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. Crashes are classified as those causing abnormal program termination, whereas benign outcomes have no change from the fault-free outcome. We found that faults belonging to the backward slice of control and pointer data have a higher chance of causing EDCs (for example, faults in loop termination conditions)[1].

3. HEURISTICS

We formulated heuristics to identify detector placement points for EDCs, on the basis of our initial study. 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 4.

We explain the heuristics with the code in Figure 2 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.

3.1. 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]The fault injection methodology, examples and complete results are presented in detail in our prior work [Thomas and Pattabiraman 2013b].

(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

```
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. 2: Example Code for Function and Data Categorization

We call such functions Optimized EDC Functions (OEF). For example in Figure 2, `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.

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.

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`. When a fault occurs in the OEF, the return value 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`.

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.

The remaining functions are side-effect causing functions which do not satisfy conditions for OEFs (`conv422to444` and `store_ppm_tga`). We elaborate the heuristics applicable to such functions in the following section.

3.2. 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.

Control Data: 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. 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 increases as the amount of data affected in downstream computations within that function increases, and as the inter-procedural loop nesting level decreases.

The EDC causing nature of function terminating branches increases, as the amount of data affected downstream within that function increases. For example, a fault at `B1`, causing a branch flip to `true`, abnormally terminates function execution, thereby missing the loop computations at `B2`. Also, the function terminating branch `B1` has a loop level of 1, since the function `conv422to444` is called within a loop at `C3`. 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`.

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.

[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.

(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. Our fault injection experiments (in Section 2) show 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 [Hari et al. 2012], 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. 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.

4. 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 3.

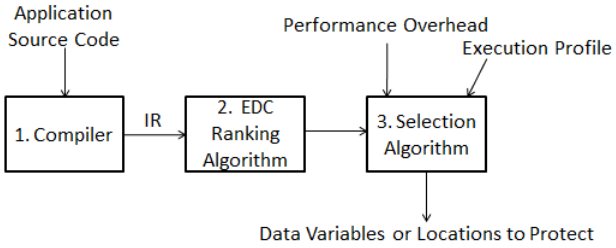

Fig. 3: Technique Workflow with required inputs

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 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.

Table I: Attribute Values and Static EDC Rank for data items from example in Figure 2. Static EDC rank is calculated using equation 1 with the values of α = 4, β = 3, μ = 2, and γ = 1. Higher EDC rank implies higher likelihood of EDC

Data Item	OuterLoop Level	InnerLoop Level	DomLoop Level	Data-Within	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
C0	3	0	0	1/2	0.1667
C1	3	0	0	1/3	0.1667

The workflow of our technique is outlined in Figure 3, 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 [Cytron et al. 1991]. 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 3 in Sections 4.1 and 4.2.

4.1. EDC Ranking Algorithm

In this phase (step 2 in Figure 3), we first identify the initial dataset, i.e., the list of potential EDC causing data items based on the heuristics in Section 3. 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 3, 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 2, the EDC rank should be such that $\mathrm{B2}_{\mathrm{edcrank}} > \mathrm{B5}_{\mathrm{edcrank}} > \mathrm{P2}_{\mathrm{edcrank}}$. 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 2 are shown in Table I. 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 I 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, branch `B2` has 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 function terminating condition `B1` has value 2 in Table I.

(4) *DataWithin* - The amount of data affected by the data item. This applies to OEF calls, branches that are not loop or function terminating, 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 I.

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:

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

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 in our prior work [Thomas and Pattabiraman 2013a].

4.2. Selection Algorithm

In this phase, i.e., step 3 in Figure 3, 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 5).

We model the problem of selecting the EDC data items as the 0-1 knapsack problem [Cormen et al. 2001]. 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:

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

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

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

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. 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 4. 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 4. 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 4.1). After calculating the weighted EDC ranks for all the data items, the final set of EDC detector locations is computed using equation 3.

5. EXPERIMENTAL SETUP

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

Implementation: We implemented the EDC ranking and the selection algorithm (steps 2 and 3 in Figure 3) 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,

```
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. 4: 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

and (b) instrumented to place detectors identified using profile data[3] under the given performance overhead bound. Third, the instrumented IR is compiled into machine code using the LLVM compiler.

We used *funcrange* of 5 in our experiments based on coverage results obtained by varying its value (see [Thomas and Pattabiraman 2013a] for details). 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 simulate our detectors by instrumenting the IR with trace calls at the locations chosen for detector placement. 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.

We simulate *ideal detectors*, as we do not consider reaching stores for loads, and function pointers when computing the backward slice. Hence, the coverage may be lower with actual detectors based on this backward slice. Also, since we approximate the backward slice, the performance overhead does not reflect the actual overhead of

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

these ideal detectors. We compare the coverage results of the actual detectors under the actual performance overhead, versus our ideal detectors in Section 6.2.

Table II: 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	test-img.jpg	PSNR between faulty and fault-free decoded images(30)
MPEG2 (9832)	Video Decoder	mei-16v2.m2v	PSNR between faulty and fault-free decoded image set (30)

Benchmarks: We use four applications from Parsec [Bienia et al. 2008], and two from Mediabench [Lee et al. 1997] 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 [Cong and Gururaj 2011; Li and Yeung 2007; Misailovic et al. 2010; Sundaram et al. 2008]. The benchmark characteristics are explained in Table II.

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 these benchmarks, compared to the inputs in Table II. We observed a 3% variation in EDC coverage for JPEG with input lena.jpg. We observed less than 2% difference in coverage for BlackScholes, Swaptions and X264. The inputs used were sim-dev, sim-medium, and eledream_64x36_3.y4m respectively. We did not have other inputs for MPEG and Canneal.

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. We made small changes to Blackscholes, Canneal and Swaptions benchmark programs as explained in our prior work [Thomas and Pattabiraman 2013a].

Fidelity Metrics and Threshold Values: We use the QoS metrics from prior work [Misailovic et al. 2010] 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 II). 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 II. We use our LLVM compiler based fault injector LLFI to perform the injections, and classify the outcomes as Crash, Benign, EDC and Non-EDC as explained in Section 2. 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) [4]. 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.

6. 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 fidelity threshold values on the EDC coverage. Finally, we present a quantitative comparison between our technique and a technique proposed in prior work.

6.1. Error Outcome Rates

Table III 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 6.4).

6.2. Coverage Under Varying Performance Overheads

EDC Coverage: Figure 5 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 applica-

[4]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

Table III: 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

tions 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%).*

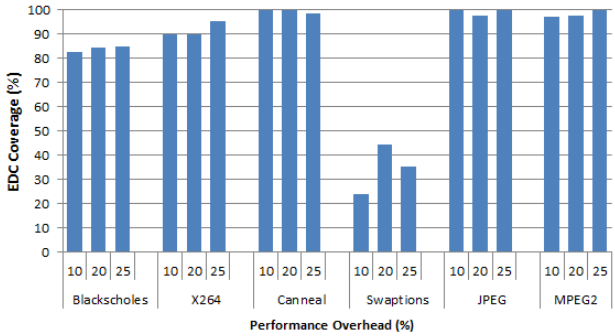

Fig. 5: EDC Coverage using our technique under performance overheads of 10%, 20% and 25%. Higher is better.

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.

Since we use simulated ideal detectors for our experiments (Section 5), we study the EDC coverage of the actual detectors under the performance overhead of 25%. The results show that Blackscholes, Swaptions and Canneal have the same coverage as our simulated detectors, with an average EDC coverage of 71% versus 72% with the ideal detectors. The remaining three benchmarks MPEG, JPEG and X264, have a coverage drop to an average of 56% because of the faults propagating to loads which initially occurred at backward slice of one of their reaching stores (flush_buffer function in MPEG and x264_cabac_encode_decision_c in X264). This can be partially resolved through a more accurate pointer analysis, which our current infrastructure does not support. However, this is not a fundamental limitation of our approach.

Non-EDC and Benign Coverage: Figure 6 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.

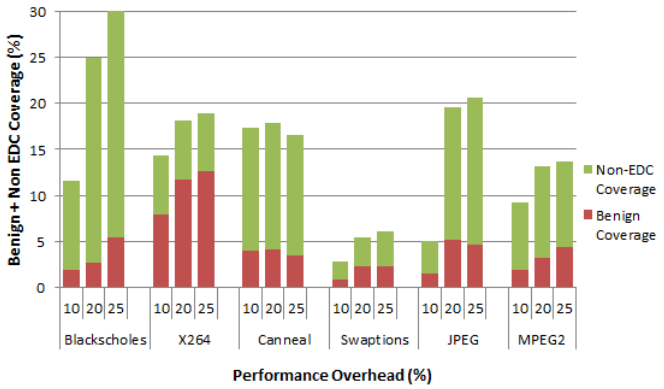

Fig. 6: Non-EDC and Benign Coverage for our technique, under performance overheads of 10%, 20% and 25%. Lower is better.

Summary: From Figures 5 and 6, 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 III, 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.*

6.3. 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. When the PSNR value was increased from 30 to a value of 40 or 50, the EDC rate increased drastically, but the additional images classified as EDCs were really Non-EDCs with very minute or no difference to the human eye.

Figure 7 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).

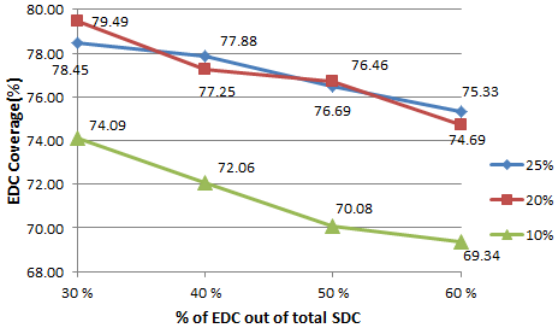

Fig. 7: Average EDC Coverage for the four Parsec benchmarks such that X% of most egregious SDCs are categorized as EDCs (under performance overheads of 10%, 20% and 25%)

6.4. Quantitative comparison with Prior Work

In this section, we quantitatively compare our technique with that of Sundaram et al. [2008] 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.

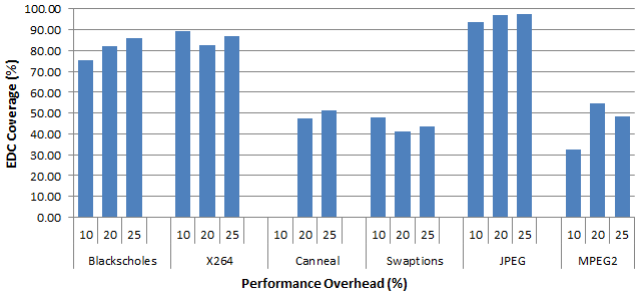

Fig. 8: EDC coverage by Selective Duplication Technique [Sundaram et al. 2008] under performance overheads of 10%, 20% and 25%

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 *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 8 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 5), 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*.

7. COMPILER OPTIMIZATIONS

In this section, we study the effect of compiler optimizations on the error resilience and vulnerability of soft computing applications. Compiler optimizations transform the application intermediate code for enhanced performance, but these transformations can lower error resilience by making some code regions more prone to EDC causing faults, compared to the unoptimized version. For example, in the Loop Invariant Code Motion (LICM) optimization, code that repeatedly assigns to the same value inside a loop, i.e., invariant code, is moved outside the loop to the loop preheader. This optimization can result in higher EDCs as a fault in the loop header variable potentially affects every iteration of the loop. However, in the unoptimized code, there are lesser chances of an EDC since the variable would be reset at every iteration of the loop.

Our technique for error detector placement incorporates heuristics which are based on static analysis of the application intermediate code. Hence, compiler optimizations have a direct effect on our detector placement technique, and we study the effect of the optimizations both with and without our technique.

Prior work [Rehman et al. 2011; Sridharan and Kaeli 2009] has focused on the effect of compiler optimization on the *vulnerability* of applications by studying their effect on microarchitectural units. However, we study error resilience (apart from vulnerability) which is a property of the application and does not depend on hardware characteristics. Second, we focus on soft computing applications, where resilience and vulnerability pertains to faults that cause the worst outcomes, rather than just crashes. Third, we study the effect of individual optimizations, rather than optimization collections. This is a step towards formulating a systematic process for grouping optimizations according to different resilience levels.

7.1. Optimizations Considered

Compiler optimizations can affect the error resilience of the program in three ways: (1) dichotomous behaviour on baseline resilience, which makes it difficult to gauge the effect of the optimization, (2) Change in heuristics, which affects the detector placement locations, and hence the EDC coverage of the technique (3) Reduction in static and dynamic code size (i.e., number of instructions executed at runtime), making certain regions of code more susceptible to faults. The optimizations and their effects are enumerated in Table IV.

7.2. Methodology

We evaluate our technique by performing fault injection on the benchmark programs in Table II, using LLFI. These fault injection experiments are performed on fifteen versions of each benchmark, namely the unoptimized code, and the fourteen optimized versions. For now, we consider only one optimization at a time. All the experiments are run with the `O0` option. We keep the fidelity threshold value and number of injections the same across all versions.

We inject faults and measure the EDC coverage of our technique under the fifteen versions of each benchmark. We inject faults one at a time in each version, and keep the fidelity threshold value and number of injections the same across all versions. We injected 5000 faults for each benchmark and optimization combination, for a total of $75,000$ faults per benchmark. The EDC rates are statistically significant within an error bar of $\pm$ 0.8% at the 95% confidence interval (Table V).

Baseline resilience: We first study the effect of the compiler optimizations on the baseline resilience by analyzing how the EDC rates (i.e., the percentage of EDCs out of the total number of faults injected) vary between the unoptimized and optimized version. When the EDC rates are within the error bars, they are considered to be the

Table IV: Description of compiler optimizations studied along with their behaviour and effects

Optimizations	**Description**	**Effect**
Combine Redundant Instructions, Common Subexpression Elimination (CSE) and Global Value Numbering (GVN)	replaces redundant code or expressions with single instruction or variable	dichotomous behaviour - higher likelihood of EDC when fault strikes combined instruction versus original code, but lower likelihood of fault striking the former
Loop Invariant Code Motion (LICM)	moves invariant code within the loop to the loop pre-header	same conflicting effect as above, when fault strikes invariant code outside loop versus that inside the loop.
Loop-Unswitch	moves branches within loops to outside the loop bodies, making these branches loop preheaders	same conflicting effect on branch with loops in the branch body versus branch within loop body (unoptimized code).
Sparse Conditional Constant Propagation (SCCP) and Inter-Procedural SCCP (IP-SCCP)	propagate constants through code, making certain conditional branches that use these constants unconditional	branches removed (heuristics `H2`, `H3` and `H4` may change)
Function Inlining	inlines chosen functions	heuristic for most side-effect free function calls ineffective (`H1`).
Loop Unroll	unrolls loops based on unrolling factor	dynamic number of branches reduces (`H2`, `H3` and `H4` heuristics affected)
Loop Reduce	performs strength reduction of array indices within loops - adds new instructions and merge nodes (phi)	better heuristics at new instructions/locations.
Jump Threading	identifies distinct threads of control flow running through a basic block, eliminates redundant branches	placement locations for branches might change (heuristics `H2`, `H3` and `H4`)
Aggresive Dead Code Elimination (ADCE)	removes dead code from program	Reduction in static and dynamic code
Loop Simplification (LoopSimplify)	simplifies code within loop	same as above
Merge-return	unifies exit blocks in function	same as above

same between optimized and unoptimized. When the rates vary beyond the error bars, we perform the two-sample hypothesis test (z test) to determine if they are different. The null hypothesis is that the EDC rates are the same between the optimized and unoptimized levels. We favour the alternative hypothesis over the null hypothesis, i.e., we consider the EDC rates to be different, when the p-value is less than 0.1 (we measure both the 0.05 and 0.1 levels).

Resilience with detection technique: We study the effect of the optimizations on the detection technique by analyzing the change in EDC coverage of the detection technique before and after each optimization, under the performance overhead bound of 20% of the dynamic code size. Higher EDC coverage implies better resilience. We measure the EDC coverage of the technique for each of the fifteen versions of a benchmark.

We consider an optimization safe under the technique, if the EDC coverage is comparable to that of the unoptimized code, i.e., higher or within 5% lower with respect to the unoptimized code. When the optimization is unsafe, we qualitatively examine the detector locations chosen in the optimized version to understand why the EDC coverage is lower.

7.3. Results

In this section, we present the baseline EDC rates (i.e., without the detection technique) and the EDC coverage of our technique for each benchmark application, under the fourteen optimizations.

Effect on Baseline Resilience Table V shows the EDC rates across the benchmarks for the unoptimized version, and the fourteen optimized versions. The last row represents the error bars of the EDC rates for the respective benchmarks. When the optimizations varied from the unoptimized version beyond the error bars, we indicate the direction of the variation and the p-value levels (0.1 or 0.05).

From Table V, the optimizations Function Inline, Loop-Unswitch and LoopSimplify have EDC rates within the error bars, compared to the unoptimized versions. Therefore, we do not consider these optimizations further. In the remaining optimizations (except LICM), benchmarks that have a significant reduction in baseline resilience also have a large reduction in dynamic code size compared to other benchmarks. Intuitively, *a large reduction in dynamic code size implies that a fault affecting the application has a more pronounced effect, i.e., higher likelihood of causing an EDC.* We investigate the optimizations that significantly lower the baseline resilience (higher EDC rates), based on the p-values[5].

Table V: EDC rates in each benchmark under the fourteen optimizations, and the unoptimized version. A lower EDC rate is better. When the rate is beyond error bars, we add H or L (implies higher or lower). * implies p-value <= 0.05 and ** implies p-value <= 0.1

Optimizations	**Black-scholes**	**X264**	**Canneal**	**Swaptions**	**JPEG**	**MPEG**
InstCombine	9.9	2.48	4.56 H*	2.5	3.68	2
LICM	9.48	2.96 H**	3.26	3.12 H**	3.56	2.1
SCCP	9.38	2.1	3.94 H	2.98 H	4.08	1.7
IP-SCCP	10.48	2.28	3.92 H	2.98 H	4.22	2.04
FunctionInline	8.68 L**	2.3	3.86	2.44	3.86	1.74
Loop-Unswitch	10.36	2.34	3.22	3.28	3.66	1.9
LoopReduce	9.14 L	2.7 H	3.28	3.0	3.82	2.18
LoopUnroll	8.48 L*	2.82 H	3.26	2.44	4.16	2.56
Jump-Threading	10.92 H	2.7 H	3.32	2.76	3.8	2.38
ADCE	9.28	2.22	3.98 H	2.72	3.82	2.14
CSE	13.94 H*	2.02	4.04 H	2.76	3.7	2.02
LoopSimplify	9.1 L	2.4	3.18	2.24	3.6	1.92
Merge-return	11.4 H**	2.52	3.04	2.58	4.12	1.96
GVN	11.02 H	2.3	3.56	3.2 H**	3.66	2.14
UnOpt	10	2.24	3.32	2.36	3.76	2.3
Error Bars	**0.8**	**0.4**	**0.49**	**0.42**	**0.53**	**0.66**

[5]A lower p-value implies that we can reject the null hypothesis with more certainty

Inst-Combine, CSE, Merge-return and GVN: In all four optimizations, *the lowering in baseline resilience occurs in benchmarks if and only if these optimizations cause significant reduction in dynamic code size compared to unoptimized code.* For example, in Inst-Combine only Canneal has a higher EDC rate compared to its unoptimized version. The dynamic code size of the optimized version of Canneal is around 25% lower than the unoptimized version, while the difference in dynamic code size is within 1% for all other benchmarks.

GVN has a lowered baseline resilience for Swaptions, which has around 19% reduction in dynamic code size. Similarly, CSE and Merge-return optimized Blackscholes benchmark have lowered resilience compared to the same optimized versions of other benchmarks.

LICM: All benchmarks except Swaptions and X264 have LICM optimized EDC rate similar to that of the unoptimized version. For X264, higher number of faults affecting the code regions within the function `cabac_encode_decision_c`, which lead to EDCs. Swaptions contains many instances of invariant code within loops, which are moved outside the loop in the LICM optimized version. Figure 9 shows one such function, `HJM_SimPath_Forward_Blocking`, which contains multiple instances of invariant code such as `iN-1` at line 4 and 7, and `BLOCKSIZE *j + b` at line 8. In the unoptimized version, faults affect these invariants (the probability of fault strike is much higher within the loop), and many of them lead to benign outcomes or Non-EDCs. This is because the invariant code is recalculated in each iteration. In the LICM optimized version for this function, faults did not affect the invariant code moved outside the loop.

```
1  void HJM_SimPath_Forward_Blocking(...){
2  ...
3     for(int b=0; b<BLOCKSIZE; b++)
4       for(j=0; j<=iN-1;j++){
5          ppdHJMPath[0][BLOCKSIZE*j +b] = pdFoward[j];
6
7          for(i=1; i<=iN-1;++i)
8             ppdHJMPath[i][BLOCKSIZE*j + b]=0;
9             //Loop Invariant: BLOCKSIZE*j + b
10      }
11    ...
12 }
```

Fig. 9: Loop Invariant Code Example in Swaptions

The increase in EDC rate for Swaptions compared to its unoptimized version, is due to faults affecting loop termination conditions whose loop body contains a call to the function `RanUnif()`. These faults lead to EDCs [6]. Higher number of faults affect these loop termination conditions because the invariant code is moved outside the loop body, leading to reduction in dynamic code size within the loop body. Further, we found that *if a random hardware fault occurs in the application, there is a low likelihood of the fault affecting invariant code that is hoisted outside the loop in LICM optimized code*.

Effect on detection technique We present the effect of the fourteen optimizations on the EDC coverage of the technique, and analyze which optimizations are *safe* under the detection technique (beyond the change in baseline resilience). Table VI shows the EDC coverage with respect to the baseline coverage (normalized at 100%) for the fourteen optimizations. The baseline EDC coverage is the coverage for the unoptimized version of the benchmark (last row in Table VI). We categorize the optimizations into resilience packages `R1` and `R2`, based on the resilience guarantees they provide.

[6] We did not observe faults affecting the invariant code outside the loop.

Table VI: EDC coverage under the fourteen optimizations with respect to the baseline EDC coverage (normalized at 100%) for each benchmark. The *unsafe* optimizations are highlighted

Optimizations	**Blackscholes**	**X264**	**Canneal**	**Swaptions**	**JPEG**	**MPEG**
InstCombine	98.88	93.97	105.6	87.89	100.46	99
LICM	99.05	99.37	108.84	101.73	100.31	98.1
SCCP	98.85	92.66	100.99	99.68	100.91	97.65
IP-SCCP	91.16	105.19	98.27	105.14	102.54	99.02
FunctionInline	98.71	103.29	101.77	101.72	101.76	98.85
Loop-Unswitch	87.24	113.13	105.47	101.72	103.31	98.95
LoopReduce	94.26	98.89	96.58	203.45	97.33	98.17
LoopUnroll	88.3	112.33	95.63	91.71	100.48	100
Jump-Threading	88.99	107.26	100	104.67	103.37	99.16
ADCE	98.15	100.9	101.88	95.74	101.18	99.06
CSE	97.33	113.13	99.12	110.57	100.49	99.01
LoopSimplify	95.7	100.88	98.82	89.01	101.53	96.88
Merge-return	94.9	108.64	105.88	78.86	104.01	100
GVN	97.77	99.36	96.82	87.74	102.16	100
UnOpt	**85.2**	**88.39**	**78.92**	**49.15**	**95.21**	**100**

Resilience Package R1 contains the optimizations that are *safe* under the technique for all benchmarks, i.e., the coverage is either higher than or no more than 5% lower than the baseline coverage. These include optimizations LICM, function inline, ADCE and CSE.

Resilience Package R2 contains optimizations that provide a lower guarantee of resilience, i.e., they are safe for five out of six benchmarks. *The R2 resilience package includes six optimizations* namely, IP-SCCP, Loop-Unswitch, Jump-Threading, Loop-InstSimplify, Merge-Return and GVN.

IP-SCCP, Loop-Unswitch and Jump-Threading are safe for all benchmarks except Blackscholes. They insert loop pre-headers, and an extra branch before the loop. In function `bs_thread` of Blackscholes, a large number of faults affect the loop termination condition of the loop which runs the monte carlo trials and calls the function which calculates the option prices (`BlkSchlsEqEuroNoDiv`). This leads to EDCs, both in the unoptimized and the optimized version. In the optimized version, a new branch condition is added to the loop preheader, and this condition is chosen for detector placement instead of the loop termination condition. This new location chosen in the optimized code does not contribute to the EDC coverage. Further, in the original code, the loop termination condition is chosen by the algorithm, and it detects the EDCs caused at that location.

Similarly, optimizations Loop-InstSimplify, Merge-Return and GVN are safe for all benchmarks except Swaptions. In Merge-Return, although there is no change in the dynamic code size compared to Swaptions unoptimized, higher number of faults affect the code region in `HJM_Swaptions_Blocking` function of Swaptions. This function performs the main calculation of the Monte-Carlo simulations (10% of total EDC in mergereturn optimized versus 5% in unoptimized). In GVN and Loop-InstSimplify, there are higher number of faults affecting the `RanUnif()` function of Swaptions, compared to the unoptimized code.

Note that all optimizations either decrease or slightly increase the error resilience of the detection technique. The one exception is Loop-Reduce, which significantly in-

creases the EDC coverage of the technique for Swaptions to 100% from 49% for the unoptimized version. This is due to the technique protecting a particular branch instruction that is highly SDC-prone, when Loop-Reduce is run on it.

Summary: We find that ten of the fourteen optimizations are safe for atleast 5 of the 6 benchmarks, of which four of them are safe for all benchmarks. This shows that *compiler optimizations do not significantly affect the error resilience provided by the detection technique for most applications*.

7.4. Vulnerability Study

Vulnerability in the context of soft computing applications, is the unconditional probability of a fault striking the application and leading to an EDC. It is the product of the EDC rate and the execution time (dynamic instruction count). Hence, while an optimization may increase a program's EDC rate, it may decrease its overall vulnerability if the performance improvement provided by it outweighs the increased EDC rate.

Table VII: The overall vulnerability of the fourteen optimizations with and without our technique (averaged across the six benchmarks) Lower vulnerability is better. Highlighted rows indicate optimizations with lower vulnerability compared to unoptimized version

Optimizations	**Baseline**			**With our Technique**			**Ratio** (x/y)
	Inst Count in millions (IB)	*EDC Rate (E)*	*Vulnerability (x = IB * E)*	*Inst Count in millions (IT)*	*EDC Rate (U)*	*Vulnerability (y = IT * U)*	
InstCombine	215.87	4.19	552.59	259.04	0.72	272.44	2.03
LICM	180.37	4.08	558.66	216.44	0.68	242.37	2.3
SCCP	216.86	4.03	636.21	260.23	0.73	291.74	2.18
IP-SCCP	215.18	4.32	641.03	258.21	0.82	277.76	2.31
FunctionInline	197.98	3.81	487.59	237.59	0.62	212.25	2.29
LoopUnswitch	179.45	4.13	576.38	215.34	0.82	257.62	2.24
Loop-Reduce	184.58	4.02	556.4	221.49	0.54	7.9	70.38
Loop-Unroll	201.79	3.95	516.22	242.15	0.82	260.82	1.98
Jump-Threading	212.59	4.31	601.49	255.11	0.82	257.64	2.33
ADCE	201.19	4.03	554.9	241.43	0.67	253.58	2.19
CSE	180.02	4.74	511.36	216.03	0.76	200.73	2.55
LoopSimplify	200.63	3.74	464.88	240.76	0.68	223.16	2.08
Merge-return	216.29	4.27	569.33	259.55	0.73	304.4	1.87
GVN	179.68	4.31	573.45	215.62	0.81	283.89	2.02
UnOpt	**217.97**	**3.99**	**538.68**	**261.56**	**0.64**	**231.62**	**2.32**

Table VII shows the vulnerability, for each of the fourteen optimizations and the unoptimized code, averaged across the six benchmarks. Column 4 shows the baseline vulnerability, while column 7 shows the average vulnerability of applications protected by our detectors, i.e., the product of dynamic instruction count and the undetected EDC rate. The EDC rate with our technique, is the fraction of undetected EDCs out of the total set of injected faults. Note that a lower vulnerability is better.

Baseline Vulnerability versus vulnerability with detection technique: Column 8 in Table VII (ratio) shows the reduction in vulnerability of our technique com-

pared to baseline vulnerability. *The vulnerability of applications with our detectors (column 7) is lower than the baseline vulnerability (column 4) for all optimizations.* Twelve optimizations out of fourteen have atleast a $2x$ reduction in vulnerability with our technique. This is because even though the dynamic count increases by 20% with our detectors, the reduction in EDC rate outweighs this increase by a much higher extent. Note that Loop-Reduce is an anomalous case as explained in Section 7.3.

Vulnerability analysis with our detectors: From column 7 in Table VII, the optimizations FunctionInline, Loop-reduce, CSE and Loop-Simplify have lower average vulnerability than the unoptimized version. Loop-Simplify has a lower resilience with our technique, i.e., it belongs to the resilience package `R2`, but it performs better in terms of vulnerability.

Baseline vulnerability analysis: When comparing the baseline vulnerabilities across optimizations (column 4 in Table VII), the optimizations FunctionInline, Loop-Unroll, CSE, and Loop-Simplify have a lower vulnerability compared to the unoptimized version. CSE has a much higher EDC rate compared to the unoptimized version, but the reduction in dynamic code size (compared to the unoptimized code) outweighs this higher EDC rate. Hence, the probability of a fault affecting the application is lower than the unoptimized version, which inturn makes the vulnerability lower than the unoptimized version.

8. RELATED WORK

We classify related work into two areas, (1) identifying critical variables, and (2) placing error detectors in a program.

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.

Cong and Gururaj [2011] focus on identifying all critical variables in an application that can tolerate deviations in output. Similar to our work, they also consider outcomes that cause large deviations from the correct output i.e., EDCs. They 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 also uses full duplication to ensure numerically accurate outputs, based on the decision taken by the runtime monitoring mechanism. 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 [Leeke and Jhumka 2010]. A critical variable, in this case, is based on its spatial and temporal impact, with respect to other software components. However, this technique uses the failure rate of the variables in deciding if a variable is critical, which requires programmer knowledge and manual effort.

Khudia et al. [2012] 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. 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. [2012] address the problem of detector placement and 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, 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 unclear how representative are these functions of general programs. Third, for the 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. [2005] develop a set of heuristics for strategic placement of detectors to detect crash causing errors with low detection latency. While the heuritics help in placing detectors to preemptively detect crashes, their coverage for SDC (and EDC) causing errors is low. Further, their approach requires constructing the DDG apriori, which has high performance overheads.

Snap [Carbin and Rinard 2010] 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. While similar to our work on using program characteristics to identify detector placement points, their technique requires fuzzing, which is analogous to fault injection, and is 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 [Zhang et al. 2010], in which the average performance overhead is reduced to 38%. However, the focus is on detecting SDCs with high coverage, which can cause DAFT to incur high Non-EDC coverage.

9. CONCLUSION

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 work, we develop heuristics for identifying EDC-prone regions of code and data in the application, and devise 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.

ACKNOWLEDGMENTS

We thank the reviewers of DSN and this journal, for their comments that helped improve the paper. We also thank Sasa Misailovic for help with the x264 benchmark, and Jiesheng Wei and Sathish Gopalakrishnan 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.

REFERENCES

W. Baek and T. M. Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation *(PLDI)*.

C. Bienia, S. Kumar, J.P. Singh, and K. Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications *(PACT)*. 72–81.

M. Carbin, S. Misailovic, and M. Rinard. 2013. Rely: Verifying quantitative reliability for programs that execute on unreliable hardware *(OOPSLA'13)*. 33 – 52.
M. Carbin and M. Rinard. 2010. Automatically identifying critical input regions and code in applications *(ISSTA)*. 37–48.
Nicholas P. Carter, Helia Naeimi, and Donald S. Gardner. 2010. Design Techniques for Cross-layer Resilience *(DATE)*. 1023–1028.
J. Cong and K. Gururaj. 2011. Assuring application-level correctness against soft errors *(ICCAD)*. 150 –157.
T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. 2001. *Introduction to Algorithms*.
R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS* 13, 4 (1991), 451–490.
M. De Kruijf, S. Nomura, and K. Sankaralingam. 2010. Relax: an Architectural Framework for Software Recovery of Hardware Faults *(ISCA)*. 497–508.
P. Dubey. 2005. Recognition, mining and synthesis moves computers to the era of tera. *Technology@ Intel Magazine* (2005), 1–10.
J.E. Fritts, F.W. Steiling, and J.A. Tucek. 2005. MediaBench II video: expediting the next generation of video systems research. *SPIE - Embedded Processors for Multimedia and Communications II* (2005), 79–93.
S. Hari, S. Adve, and H. Naeimi. 2012. Low-cost Program-level Detectors for Reducing Silent Data Corruptions *(DSN)*. 181–188.
M. Hiller, A. Jhumka, and N. Suri. 2002. On the Placement of Software Mechanisms for Detection of Data Errors *(DSN)*. 135–144.
D. Khudia, G. Wright, and S. Mahlke. 2012. Efficient soft error protection for commodity embedded microprocessors using profile information *(LCTES)*.
C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation *(CGO)*. 75–86.
C. Lee, M. Potkonjak, and W.H. Mangione-Smith. 1997. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems *(MICRO)*. 330–335.
M. Leeke, S. Arif, A. Jhumka, and S.S. Anand. 2011. A methodology for the generation of efficient error detection mechanisms *(DSN)*. 25–36.
M. Leeke and A. Jhumka. 2010. Towards understanding the importance of variables in dependable software *(EDCC)*.
L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra. 2010. ERSA: Error Resilient System Architecture for Probabilistic Applications *(DATE)*. 1560–1565.
Xuanhua Li and D. Yeung. 2007. Application-Level Correctness and its Impact on Fault Tolerance *(HPCA)*. 181 –192.
S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn. 2011. Flikker: saving DRAM refresh-power through critical data partitioning *(ASPLOS)*. 213–224.
S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. 2010. Quality of Service Profiling *(ICSE)*. 25–34.
S. Narayanan, J. Sartori, R. Kumar, and D. Jones. 2010. Scalable stochastic processors *(DATE)*. 335–338.
K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer. 2005. Application-based metrics for strategic placement of detectors *(PRDC)*. 8.
S. Rehman, M. Shafique, F. Kriebel, and J. Henkel. 2011. Reliable Software for Unreliable Hardware: Embedded Code Generation Aiming at Reliability *(CODES+ISSS)*. 237–246.
M. Samadi, J. Lee, D. Jamshidi, A. Hormati, and S. Mahlke. 2013. "SAGE": Self-Tuning Approximation for Graphics Engines *(MICRO-46)*. New York, NY, USA.
A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation *(PLDI '11)*. 164–174.
P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. 2002. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic *(DSN)*. 389–398.
D.P. Siewiorek. 1991. Architecture of fault-tolerant computers. *Proceedings of IEEE* (1991), 79–91.
V. Sridharan and D. Kaeli. 2009. Eliminating Microarchitectural Dependency from Architectural Vulnerability. *(HPCA)*. 117–128.
A. Sundaram, A. Aakel, D. Lockhart, D. Thaker, and D. Franklin. 2008. Efficient fault tolerance in multi-media applications through selective instruction replication *(WREFT)*. 339–346.
Anna Thomas and Karthik Pattabiraman. 2013a. Error Detector Placement for Soft Computing Applications *(DSN)*. 12.
A. Thomas and K. Pattabiraman. 2013b. LLFI: An Intermediate Code Level Fault Injector For Soft Computing Applications *(SELSE)*.
L.A. Zadeh. 1997. What is Soft Computing? *Soft computing* 1, 1 (1997), 1–1.
Y. Zhang, J. Lee, N. Johnson, and D. August. 2010. DAFT: Decoupled Acyclic Fault Tolerance *(PACT)*. 87–98.