Effect of Compiler Optimizations on the Error Resilience of Soft Computing Applications

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

ABSTRACT

While hardware errors are on the rise as chip sizes reduce, users of commodity systems expect a near faultless experience with low degradation in performance. Developers tune for higher performance by enabling compiler optimizations on code, but these optimizations affect the resilience of applications, making it difficult to maintain an error resilience guarantee when multiple optimizations are applied together (e.g., -O3 in `gcc`).

We focus on soft computing applications, (e.g., multimedia applications) that 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). We study how four specific compiler optimizations affect the resilience of soft computing applications. Further, we investigate how the optimizations affect the detector placement locations for detecting EDC causing faults. This helps us identify *safe* compiler optimizations that maintain a certain guarantee on the error resilience of the application. Our work is a first step towards identifying the performance-resilience tradeoff space.

Categories and Subject Descriptors

D.3.4 [**Software Engineering**]: Processors

General Terms

Reliability, performance

1. INTRODUCTION

Transient faults are on the rise with reducing chip sizes and transistors operating at lower voltages [13]. However, users desire a near faultless experience when performing their daily tasks, while at the same time demanding high performance. Achieving both these goals is challenging in the face of transient faults. Hence, developers and architects are consciously making decisions regarding the trade-off between performance and reliability of systems they build. One such tradeoff arises when deciding on whether to enable compiler optimizations, which while increasing application performance, may lower its resilience. We explore this tradeoff in this paper.

We focus on a class of applications called soft computing applications [19]. These applications can tolerate most deviations in outcomes while producing acceptable outputs. Examples of soft computing applications are multimedia decoding applications, which can tolerate blurry decoded images, and machine learning applications, which can tolerate noise. Researchers have predicted that future workloads will belong primarily to this category [3]. Soft computing applications have an associated fidelity metric, which is a quantitative measure of the output quality. We use the term *Egregious Data Corruptions (EDCs)* to denote outcomes that deviate significantly from the fidelity metric, i.e., unacceptable outcomes.

Resilience is the property of an application to tolerate hardware faults *once they have occurred in the application*. In the context of soft computing applications, the resilience is the ability of an application to prevent an error from becoming an EDC. In prior work, we developed a technique for identifying code locations for placing error detectors in soft-computing applications, to detect EDCs with high coverage [16]. In this paper, we study how compiler optimizations modify these placement decisions, and hence affect the detection coverage of soft computing applications. We also investigate the effect of compiler optimizations on the baseline EDC rate of an application (without our technique).

Compiler optimizations may 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 which is assigned repeatedly 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 a higher EDC rate as the effect of a fault in the loop header variable 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.

Prior work has focused on the effect of compiler optimizations on application vulnerability [14, 2] (see Section 2). However, these studies do not investigate how optimizations affect the application resilience, which is different from vulnerability. Other work [12, 5, 8] has investigated the problem of optimal error detector placement for different failure types. However, these studies do not investigate how their detector placement decisions are affected by compiler optimizations. *To our knowledge, ours is the first work to consider the effect of compiler optimizations on application error resilience, both with and without our detector placement technique.*

We make the following contributions in this work:

1. We study the effect of four compiler optimizations on the resilience of soft computing applications, by performing fault

injection experiments on the optimized and unoptimized versions of these applications (with no detection technique),

2. We then investigate how the optimizations modify the EDC coverage of our detector placement algorithm, by studying its coverage with and without the optimization,

3. We identify *safe* optimizations, i.e., those for which our algorithm maintained similar EDC coverages compared to the unoptimized version,

4. Finally, we develop insights on how certain application characteristics contribute to the change in resilience both with and without our detection technique.

Our work helps to better understand the coupling of compiler optimizations and reliability. Our study is a first step towards identifying compiler optimizations that maintain or achieve higher resilience compared to the baseline unoptimized versions of applications.

2. RELATED WORK

Prior work has focused on the effect of compiler optimization on the *vulnerability* of applications, i.e., the probability that a hardware fault affects the application and leads to an application failure. This is different from resilience, which is the conditional probability of a failure given an error in the application. Unlike vulnerability, resilience is a property of the application alone, and does not depend on hardware characteristics.

Sridharan and Kaeli [14] developed the Program Vulnerability Factor (PVF), a methodology to estimate the vulnerability of an application at the program level. PVF abstracts at the program level the Architectural Vulnerability Factor (AVF), which is an aggregate metric that determines whether a hardware fault would escalate to an erroneous output. AVF considers the occupancy of a microarchitectural unit in determining whether a fault in that unit is likely to lead to an application failure, and is hence intricately tied to the microarchitecture. PVF can estimate vulnerability under differing microarchitectures, while having a strong correlation with the AVF. Sridharan et al. use the PVF metric to explore the effect of compiler optimizations on the `bzip` application.

Demertzi et. al. [2] study the effect of compiler optimizations on vulnerability at the microarchitectural level. They apply a number of `gcc` optimizations to SPEC benchmarks, and analyze the resulting machine code output. Optimizations improve the code performance, allowing the code to be *in flight* for a lesser period of time, and hence making it less susceptible to hardware faults. They determine the susceptibility of the code to a fault, using a metric called Expected Failures during execution, which takes into account the AVF and the number of Failures per 1 billion operations (FIT). They then explore the effect of optimization collections (four optimization levels in `gcc`) on specific microarchitectural units such as the Reorder Buffer, the Instruction Fetch Queue, and the Load Store Queue.

In addition to the focus on resilience, our study differs from the above ones in that it considers the effect of individual optimizations (e.g., Loop Invariant Code Motion) on the resilience of the application. On the other hand, the above studies look at the effects of collections of optimizations enabled by the optimization levels in gcc (e.g.,-O2), which makes it difficult to pinpoint which optimization is responsible for the reduction in resilience (if any).

3. BACKGROUND

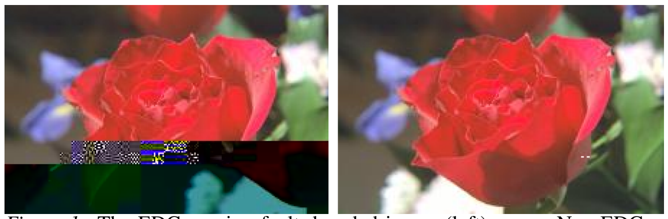


Figure 1: The EDC causing fault decoded image (left) versus Non-EDC causing fault decoded image (right) from the JPEG decoder

In this section, we present our fault model, the concept of an EDC, and the high-level idea of our previous work that identifies error detector locations for EDC causing faults.

Fault Model: We consider transient hardware faults that occur in the processor, specifically those 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 [11]. As in prior work, we do not consider faults in floating point registers [4] - this is part of future work.

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 [9]. 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 either an EDC or a Non-EDC, depending on the fidelity threshold value of the outcome.

The example in Figure 1 shows the faulty decoded images of the JPEG decoder (part of Mediabench [7]), when a fault is injected into the program. The fidelity threshold is the 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 (not shown).

Detector Placement Technique: In our prior work [16], we developed a technique to identify detector placement locations to detect EDCs with high coverage while efficiently differentiating them from the Non-EDC and benign faults. This technique relies entirely on static analysis of the code, and the dynamic execution profile of the application. Other work [4, 15] has identified control and pointer data as detector placement targets. However, our technique identifies the following program data as critical from EDC detector placement point of view:

1. Control data which affects large amount of data. For example, loop termination conditions, and branches with pointer arithmetic within their bodies.

2. Pointer data which points to large amount of data, and hence a fault in the lower order bits of the pointer cause an EDC. For example, faults affecting array indices usually cause a Non-EDC or crash (depending on the bit location).

3. Calls to certain functions that return a value, and do not cause any side-effects. A detector is placed to verify the return

value of the function, and no detectors are placed within the function, as it is side-effect free.

We performed an initial study to formulate heuristics that identify EDC causing data. More details may be found in [16]. The underlying common characteristic of these heuristics was that *data or instructions affecting larger amount of data, were highly likely to cause an EDC, and should be protected with detectors.* Let us consider the example in Figure 2, which is based on the MPEG benchmark, from the Mediabench Benchmark. The function `conv422to-444` 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). We found that faults in branches `B1` and `B2` cause an EDC as they can affect the entire `dst` array, but faults in the low order bits of the array index of `Clip` in `P1` or a fault in branch `B3` causes a Non-EDC, as they affect only one element in the `dst` array.

```
void conv422to444(char *src, char* dst, int width,
    int height, int offset) {
    ...
    w = width>>1;
    if(dst < src + offset)    //B1
      return;
    for(j=0; j < height; j++) {   //B2
     for(i=0; i < width; i++) {
          i2 = i<<1;
          im1 = (i < 1) ? 0 : i-1;  //B3
          ...
          dst[i2] = Clip[(21*src[im1])>>8];  //P1
     }
    }
    ...
}
```

Figure 2: Example Code of EDC versus Non-EDC data

Further, based on the heuristics, we developed an algorithm to identify program locations for placing error detectors. The goal of the algorithm is to preemptively detect EDC causing faults in soft computing applications, under a given performance overhead that the user is willing to tolerate. The algorithm 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 (contains the dynamic instruction counts of the application).

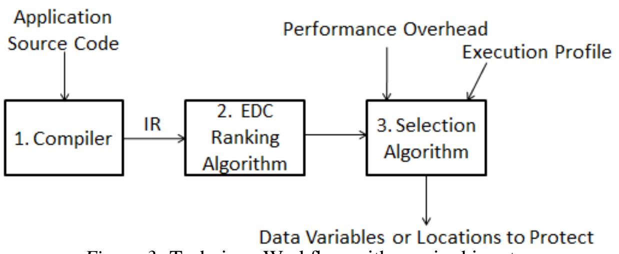


Figure 3: Technique Workflow with required inputs

The workflow of our technique is outlined in Figure 3, and consists of three steps. First, we compile the application source code into an Intermediate Representation (IR). Second, we rank the above critical instructions according to their EDC causing nature, based on the heuristics and static analysis of the code. 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.

Implementation: We implemented the EDC ranking and the selection algorithm as custom passes in the LLVM compiler version 2.9 [6]. First, the application source code is compiled into the LLVM Intermediate Representation (IR) along with the `mem2reg` optimization (i.e., promote loads/stores to registers). Second, the IR is (a) statically analyzed to compute the static EDC rank for the EDC causing instructions, and (b) instrumented to place detectors identified using profile data, under the given performance overhead bound. We wrote a custom pass for obtaining profile data and for measuring the performance overhead, using LLVM basic block profiling pass. Third, the instrumented IR is compiled into machine code using the LLVM compiler's code generation pass.

The error detectors are derived by replicating the static interprocedural 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. 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 IR on the same input, with no faults injected.

In this paper, the compiler optimizations are done at the intermediate code level, and we study how the EDC coverage of our technique varies under these optimizations[1]. At a high level, this helps us in understanding how the detection technique is affected by the compiler optimizations. As we show later, *certain optimizations reduce the EDC coverage of our technique, and the extent of reduction depends on program characteristics.*

4. OPTIMIZATIONS

In this section, we present the specific optimizations under which we evaluate the resilience of soft computing applications, and the rationale behind choosing these optimizations for our study. For each optimization, we formulate research question(s) on the behaviour of faults in applications under the optimization. Note that the research questions are orthogonal to our detection technique. We investigate these research questions in Section 6.1. They examine how faults affecting the applications behave under these optimizations, with no detection technique in place. We study the effect of the optimizations on our technique in Section 6.2.

We define EDC rate as the fraction of EDC outcomes out of all fault outcomes of the application. The optimizations affect the baseline resilience of the application by either (a) increasing the EDC rate compared to unoptimized code, i.e., lowering the resilience, or (b) decreasing the EDC rate compared to unoptimized code, i.e., improving the resilience of optimized code.

4.1 Combine Redundant Instructions: Inst-Combine

The Combine Redundant Instructions optimization (*Inst-Combine*), as the name implies, optimizes code by combining multiple instructions into one. The question we ask for this optimization is:

RQ1: Do faults affecting inst-combined optimized code have a higher likelihood of causing EDCs, thereby increasing the EDC rate compared to the unoptimized version?

There are two conflicting answers to this question:

- A hardware fault affecting the combined instruction in the optimized version has a higher likelihood of leading to an EDC compared to a fault affecting the redundant instructions in the unoptimized version. This is because the combined

[1]We had only enabled the `mem2reg` optimization of LLVM in our earlier work [16].

instruction represents the collection of original redundant instructions, and might have a more pronounced effect on the output, thus leading to an EDC.

- The probability of the combined instruction in the optimized version being affected by a fault, is lower than the set of redundant instructions in the unoptimized version, as it has a smaller footprint.

4.2 Loop Invariant Code Motion: LICM

Loop invariant code motion (*LICM*) is an optimization technique which moves invariant code within the loop to the loop pre-header, i.e., it is no longer executed within loop iterations. This optimization is prevalent in popular compilers such as `gcc`, and LLVM. Note that the effect of the optimization on the code is highly dependent on the application. In particular, it works in applications that contain invariant code within loops that can be moved outside. The question we ask around this optimization is:

RQ2: Do faults affecting the invariant code moved outside the loop in LICM optimized application have a higher likelihood of causing EDCs, thereby increasing the EDC rate compared to the unoptimized version?

Similar to Inst-Combine, LICM also brings forth contrasting possibilities in terms of soft errors:

- There is a lower likelihood of a fault affecting the invariant code that is moved to the preheader (executed less frequently than code within the loop), and hence lower likelihood of EDCs.

- On the other hand, if a fault affects the invariant code moved outside the loop, there might be a greater propagation of the error since the value is not being reset in each iteration as in the unoptimized version. This leads to a higher likelihood of EDCs.

4.3 Loop Unrolling: Loop-Unroll

Loop unrolling (*loop-unroll*) trades off static code size for performance, by unrolling the loops in a program (as the name suggests) [18]. This reduces branch misprediction penalties, as there are fewer branches to (mis-)predict. Also, if the loop statements are independent, higher performance gain can be achieved by loop parallelism. Note that the effect of the loop unrolling is dependent on the loop unrolling factor, and factors such as the loop trip count. The question we ask regarding this optimization is:

RQ3: Do faults affecting loop-unrolled optimized code have a lower likelihood of causing EDCs, thereby lowering the EDC rate compared to the unoptimized version?

By loop unrolling, the dynamic count of loop termination branch conditions reduces. Hence, there might be a lower probability of a hardware fault striking the branch condition, and leading to an EDC.

4.4 Sparse Conditional Constant Propagation: SCCP

The Sparse Conditional Constant Propagation optimization (*SCCP*), propagates constants throughout the code. In the process, certain conditional branches that use these constants become unconditional, which in turn propagates more constants. Also, after *SCCP*, the static code size is further reduced through dead code elimination (another optimization pass).

RQ4: Do faults affecting SCCP optimized code, have a higher likelihood of causing EDCs, thereby increasing the EDC rate compared to the unoptimized version?

Due to SCCP, the dynamic code size would be lower as conditional branches and dead code are eliminated through constant propagation. We speculate that the effect of a hardware fault would be more pronounced, and hence the fault has a higher likelihood of causing an EDC.

5. EXPERIMENTAL SETUP

In this section, we present the benchmarks and the fidelity thresholds, followed by the evaluation metrics and the approach we use to analyze the effect of the optimizations on the application error resilience (both with and without our detection technique).

Benchmarks: We use six soft-computing applications, two from MediaBench [7] and four from Parsec [1]. Table 1 shows the benchmarks, and the fidelity thresholds, along with the fidelity threshold values (mentioned in parantheses in column 4). We made small changes to some of the benchmark programs as explained in our prior work [16], to make them work with our infrastructure.

Table 1: 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)
Black-Scholes (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 (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	mei16-v2.m2v	PSNR between faulty and fault-free decoded image set (30)

Fidelity Metrics and Threshold Values: We use the Quality of Service (QOS) metrics from prior work [10] as the fidelity metrics for the Parsec benchmarks. We distinguish EDCs from Non-EDCs using the fidelity threshold value. 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 manually inspected the faulty outputs, and 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 1. These fault injection experiments are performed on five versions of each benchmark's code, namely the unoptimized code, and the four optimized versions (see Section 4). We kept the fidelity threshold value and number of injections the same between the optimized and unoptimized versions.

For performing the fault-injection experiments, we use LLFI, a program level fault-injection tool that we developed [17]. LLFI works at the LLVM compiler's intermediate representation (IR) level, and allows fault-injections to be performed at specific program points, and into specific data types. In each run, a fault, i.e., a single bit flip, is 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 is 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 classify the outcomes of the injected faults into Crash, Benign, EDCs and Non-EDCs (the hang rate is very low in our experiments). All injected faults are executed i.e., the instruction into whose output the fault was injected, is executed by the program.

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 measure the EDC coverage of our technique under the optimized versions of the six benchmarks, for each optimization, and on the unoptimized version. We present only the EDC coverage, as the Non-EDC and Benign coverage did not vary much between the optimized and unoptimized versions.

We inject 5000 faults for each benchmark and optimization combination. The EDC rates are statistically significant with an error bar of $\pm$ 0.8% at the 95% confidence interval [2]. 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. We measure the EDC coverage under the performance overhead bound of 20% (the dynamic instruction count of the replicated instructions).

Analysis approach: We measure the effect of optimization on the baseline resilience (with no detection technique) by investigating the increase or reduction in the EDC rate of the optimized version versus unoptimized version. For the second part of our analysis, i.e., studying the effect of optimization on our detection technique, we perform a quantitative analysis by comparing the EDC coverage of our technique under the optimized version versus the unoptimized version.

We consider an optimization safe under our 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. In cases where the the optimization is unsafe, we qualitatively examine the detector locations chosen in the optimized version to understand why the EDC coverage is lower.

[2]Blackscholes has an error bar of $\pm$ 0.8%, while other benchmarks have an error bar within $\pm$ 0.5%.

6. RESULTS

In this section, we present the effect of compiler optimization on the application error resilience (independent of our detection technique), and answer the research questions posed in Section 4. We then present the effect of the optimizations on our detection technique, by reporting the EDC coverage of our technique for the four optimizations under the 20% performance overhead.

6.1 Effect of compiler optimization on error resilience

Table 2 shows the percentage of EDC outcomes for each benchmark under the four optimizations. The EDC percentages presented in the table are independent of our technique, and it shows how the resilience of the applications varies from the baseline unoptimized version with no detection technique in place. Recall that *the optimization reduces the error resilience of the application compared to baseline resilience, when the EDC rate of the optimized version of the application is higher compared to the unoptimized one.*

Table 2: Percentage of EDC outcomes in each benchmark under the four optimizations, and the unoptimized version. Error bars are $\pm$ 0.8%

Benchmark	**Inst-Combine**	**LICM**	**Loop-Unroll**	**SCCP**	**UnOpt**
Blackscholes	9.9	9.48	8.48	9.38	10
X264	2.48	2.96	2.82	2.1	2.24
Canneal	4.56	3.26	3.26	3.94	3.32
Swaptions	2.5	3.12	2.44	2.98	2.36
JPEG	3.68	3.56	4.16	4.08	3.76
MPEG2	2	2.1	2.56	1.7	2.3
Average	4.19	4.08	3.95	4.03	3.99

The occurrence of EDC causing faults in the optimized code varies significantly (beyond the error bars) from the unoptimized version, depending on the benchmark. This is true for all four optimizations. We present the analysis for each optimization, by considering benchmarks whose EDC rates vary significantly from the unoptimized version.

6.1.1 Inst-Combine

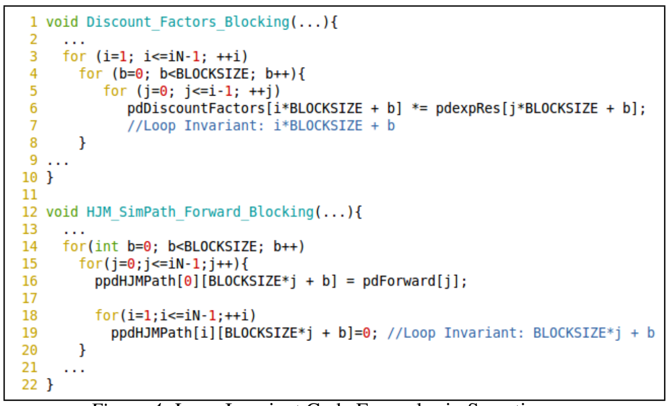
In general, Inst-Combine optimized benchmarks have a lower EDC rate than the corresponding unoptimized version, thereby maintaining the baseline error resilience of the application. However, Inst-Combine optimized Canneal has a higher EDC rate than its unoptimized version [3]. On further investigation, we found that the dynamic count of instructions in the optimized version is an order of magnitude lower than the unoptimized version. For all other benchmarks, the difference in dynamic count is within 1%. To answer `RQ1`, *faults affecting Inst-combine optimized code have a higher likelihood of causing an EDC, only if the reduction in dynamic code size with respect to the unoptimized version is highly pronounced.*

6.1.2 LICM

For LICM as shown in Table 2, all benchmarks except Swaptions and X264 have LICM optimized EDC rate similar to that of the unoptimized version. Swaptions is significantly higher than the baseline unoptimized version, i.e., the optimization lowers its resilience compared to the unoptimized version. Swaptions contains many instances of invariant code within loops. Figure 4 shows two such

[3]Swaptions and X264 also have a higher EDC rate, but they are within the error bars of their respective EDC rates.

functions, `Discount_Factors_Blocking` and `HJM_SimPath_Forward_Blocking`. The loop invariant code in `Discount_Factors_Blocking` is `i * BLOCKSIZE + b` at line 6, which is hoisted outside the loop. Similarly, for `HJM_SimPath_Forward_Blocking`, we have multiple instances of invariant code such as `iN-1` at line 15 and 18, and `BLOCKSIZE *j + b` at line 19. In the LICM optimized version for the `HJM_SimPath_Forward_Blocking` function, we did not see faults affecting the invariant code at all. In the unoptimized version, faults affect multiple invariants within the loop (the probability of the fault striking is much higher as the invariant code is within the loop), and many of them lead to benign outcomes or Non-EDCs.



```
 1 void Discount_Factors_Blocking(...){
 2   ...
 3   for (i=1; i<=iN-1; ++i)
 4     for (b=0; b<BLOCKSIZE; b++){
 5        for (j=0; j<=i-1; ++j)
 6           pdDiscountFactors[i*BLOCKSIZE + b] *= pdexpRes[j*BLOCKSIZE + b];
 7           //Loop Invariant: i*BLOCKSIZE + b
 8     }
 9 ...
10 }
11
12 void HJM_SimPath_Forward_Blocking(...){
13   ...
14   for(int b=0; b<BLOCKSIZE; b++)
15     for(j=0;j<=iN-1;j++){
16       ppdHJMPath[0][BLOCKSIZE*j + b] = pdForward[j];
17
18       for(i=1;i<=iN-1;++i)
19         ppdHJMPath[i][BLOCKSIZE*j + b]=0; //Loop Invariant: BLOCKSIZE*j + b
20     }
21   ...
22 }
```

Figure 4: Loop Invariant Code Examples in Swaptions

It is interesting to note that the increase in EDC rate for Swaptions compared to its unoptimized version, is not due to faults affecting the invariant code outside the loop (we did not observe faults affecting the invariant code). The increase is due to faults affecting loop termination conditions whose loop body contains a call to the function `RanUnif()`, and these faults lead to EDCs. These loop bodies contain invariant code similar to those in function `HJM_SimPath_Forward_Blocking`. Due to the reduction in dynamic code size of statements within the loop body of the LICM-optimized Swaptions, there are a higher number of faults affecting the loop termination conditions, and hence a higher number of EDCs.

To answer `RQ2`, the increase in EDC rate of LICM-optimized applications compared to the unoptimized version, is not due to faults affecting the invariant code hoisted outside the loop. Also, we found that *under the conditional probability that a random hardware fault occurs in the application, there is a low likelihood of a fault affecting invariant code hoisted outside the loop in LICM optimized code.*

6.1.3 Loop-Unroll

The EDC rates of most Loop-Unroll optimized benchmarks are within the error bars of their corresponding unoptimized versions. While there is an increase in the static code size for all benchmarks with this optimization, this is mostly due to the pre-requisite passes. These passes, *lcssa* (loop-closed-ssa), and *loop-simplify* insert loop pre-headers, and an extra branch before the loop. Hence, in the unoptimized version while one loop termination branch condition is present, in the optimized version two branch conditions are present for the same loop. In most benchmarks, an extra branch condition is added by the prerequisite passes, but no unrolling happens (possibly due to absence of trip count information).

To answer `RQ3`, we cannot conclude that the likelihood of faults leading to EDC for loop-unrolled optimized code is lower. Only Blackscholes has a lower EDC rate compared to the unoptimized version. We did not see any reduction in the number of faults affecting branch conditions.

6.1.4 SCCP

The EDC rates of most SCCP optimized benchmarks are similar to the corresponding unoptimized versions, except for Canneal and Swaptions. However, this increase does not have any correlation with the reduction in dynamic code size. In most cases, SCCP optimization eliminates `bitcast` instructions, and variable indices in the `getelementptr` instruction (to use constant indices).

To answer `RQ4`, certain applications have a higher likelihood of EDC causing faults in SCCP optimized code, i.e., higher EDC rates for SCCP optimized code, but this increase is not correlated with the reduction in dynamic code size. We will investigate this more in future work.

6.2 Effect of compiler optimization on detection technique

In this section, we present the effect of the optimizations on the detectors chosen by our technique. We also present the effect of the four optimizations on the EDC coverage of our technique, and which optimizations are *safe* under our detection technique (beyond the change in baseline resilience).

6.2.1 Detectors

Table 3 presents the number of detectors under different optimizations for the 20% performance overhead bound, chosen by our technique.

Table 3: Number of detectors in each benchmark under different optimizations for 20% performance overhead

Benchmark	**Inst-Combine**	**LICM**	**Loop-Unroll**	**SCCP**	**UnOpt**
Blackscholes	14	14	16	14	14
X264	330	468	301	331	331
Canneal	31	36	40	36	36
Swaptions	134	30	60	136	136
JPEG	75	63	65	61	61
MPEG2	162	167	146	165	165

Our technique identifies these locations by estimating the performance overhead of a specific location based on the dynamic count of replicated code, i.e., the backward slice of the detector location. Hence, *the detector placement locations identified using our technique is tightly coupled with how the optimizations change the benchmark code.*

For example, the number of detectors chosen by Inst-combine optimization depends on which redundant instructions are combined. The JPEG optimized version has a higher number of detectors than its unoptimized version (75 versus 61). This is because the combined instructions belong to the backward slice of the chosen detector locations, and hence the performance overhead of the detector is lower than that in the unoptimized version.

However, the number of detectors are much lower in Canneal, when Inst-Combine is applied to it. The redundant instructions that are eliminated reduce the overall dynamic count of instructions, but does not change the dynamic count of the backward slice of the detector locations. Hence, the performance overhead of the individual detectors in the optimized version is higher, and under the given performance overhead bound, lower number of detectors get chosen.

6.2.2 *EDC Coverage*

Figures 5, 6, 7 and 8 show the normalized EDC coverage with respect to the baseline coverage for the four optimizations. The baseline EDC coverage is the coverage for the unoptimized version of the benchmark. We do not show the absolute baseline coverages because our study focuses on *the change in coverage by the optimizations*. Note that the baseline coverages of all benchmarks are normalized to 100%. From Figure 6, we find that LICM is a *safe* optimization for all benchmarks under our technique.

The EDC coverage differs across benchmarks, for the four optimizations. For three of the benchmarks, JPEG, MPEG and Canneal, all four optimizations are *safe* compared to the unoptimized version, i.e., either higher than the baseline coverage or within 5% lower than the baseline. Hence, *the effect of optimization on EDC coverage depends on benchmark specific characteristics.* For the remaining three Parsec benchmarks, individual optimizations are unsafe depending on the benchmark. We study these characteristics in more detail.

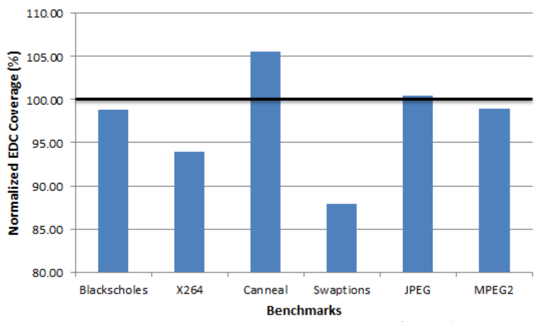


Figure 5: Normalized EDC Coverage of **InstCombine** with respect to the Unopt EDC Coverage (shown as 100%).

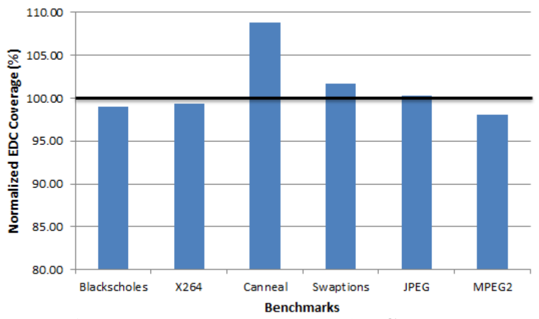


Figure 6: Normalized EDC Coverage of **LICM** with respect to the Unopt EDC Coverage (shown as 100%).

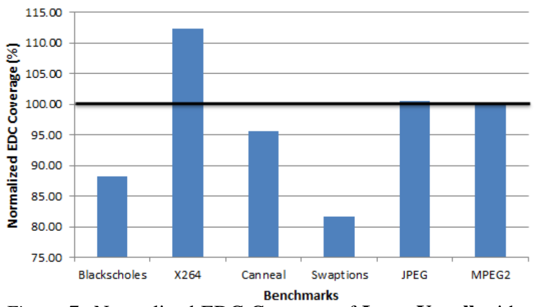


Figure 7: Normalized EDC Coverage of **Loop-Unroll** with respect to the Unopt EDC Coverage (shown as 100%).

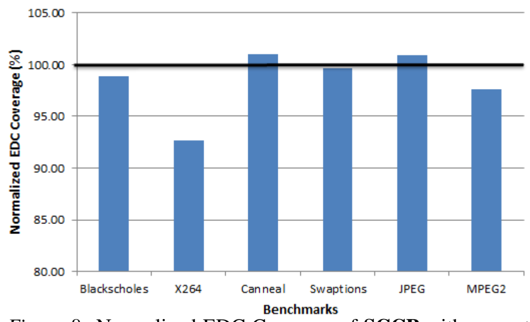


Figure 8: Normalized EDC Coverage of **SCCP** with respect to the Unopt EDC Coverage (shown as 100%).

Inst-Combine: From Figure 5, InstCombine is *safe* for all benchmarks except X264 and Swaptions. On further investigation, we noticed that for these benchmarks, *Inst-Combine modifies code such that regions that were less susceptible to faults in the unoptimized version become more susceptible in the optimized version, and this reduces the EDC coverage of our technique*. Note that these regions were not protected in the unoptimized version either, but fewer faults affected those regions compared to the optimized version, leading to fewer EDCs in the unoptimized version.

For example, in Inst-Combine optimized Swaptions, a higher number of faults affect the code within the function `RanUnif()`, compared to those affecting the unoptimized version (40% in optimized versus 22% of total EDCs in unoptimized). This function is not protected in both the optimized and unoptimized version. `RanUnif()` is responsible for the 12% reduction in coverage compared to the baseline coverage. Similarly, in X264, a higher number of faults affect code regions within the function `cabac_encode_decision_c` (14% versus 8% of the total EDCs in the unoptimized version). Detectors are not placed in this function for either the unoptimized or the optimized version. Note that both these functions have a much higher dynamic execution time compared to the remaining sections of the code. Hence, the reduction in dynamic code size results in more faults affecting these functions.

Loop-Unroll: Loop-Unroll is a safe optimization under our technique for all benchmarks except Blackscholes and Swaptions. In Blackscholes, the EDC coverage for the loop unrolled version is around 12% lower than the baseline coverage (see Figure 7). The loop termination condition at line 3 for function `bs_thread` in Figure 9, has a large number of faults affecting it. This leads to EDCs, both in the unoptimized and the optimized version. In the optimized version, while the loop is not unrolled, a new branch condition is added to the loop preheader, and this condition is chosen for detector placement while the loop termination condition is not chosen by our algorithm. In the original code, the loop termination condition is chosen by our algorithm. Hence, while this new location does not contribute to EDC coverage, the original detector on the loop termination condition would have detected the EDCs caused at that location.

```
 1 int bs_thread(void *tid_ptr){
 2         ...
 3   for (i=start; i<end; i++) {
 4         price = BlkSchlsEqEuroNoDiv( sptprice[i], strike[i],
 5                                       rate[i], volatility[i], otime[i],
 6                                       otype[i], 0);
 7         prices[i] = price;
 8         ...
 9   }
10  ...
11 }
```

Figure 9: Blackscholes code segment

The low EDC coverage in Swaptions is because of the high number of faults affecting the `RanUnif()` function (47% in optimized versus 22% of the total EDCs in the unoptimized version). *Loop-Unrolling leads to different consequences in different benchmarks, making it difficult to gauge the error resilience of applications.* This is because in cases where loops are not unrolled, the prerequisite passes add extra static branch conditions to loops, which may affect the EDC coverage.

SCCP: From Figure 8, SCCP is safe under our technique for all benchmarks except X264, under the 20% performance overhead. The low coverage in X264 is attributed to the higher number of faults affecting the `cabac_encode_decision_c` function (14% versus 8% of the total EDCs in the unoptimized version). It is interesting to note that the low coverage in the Inst-Combine optimized version of X264 is due to the same reason.

Summary: From Sections 6.1 and 6.2, we find that maintaining a resilience guarantee under compiler optimizations is entirely dependent on which regions of code become more susceptible to EDC causing faults [4]. In some cases such as LICM-optimized Swaptions and Inst-Combine optimized Canneal, the resilience of the optimized version is lower compared to its unoptimized version (in the absence of our detection technique), i.e., the EDC rate is higher compared to the unoptimized version. However, our detection technique maintains the overall resilience of these applications, because the increased number of EDC causing faults in the optimized version affects locations that are already protected by our technique. In other cases such as Inst-Combine optimized X264 and Swaptions, and SCCP optimized X264, the baseline resilience is maintained in the optimized version (the EDC rate is lower or similar), but these lower number of EDC causing faults affects those locations not protected by our technique. Hence, *our detection technique maintains the overall resilience of most applications under the three compiler optimizations (except Loop-Unroll), regardless of how these optimizations affect the baseline resilience*.

7. CONCLUSION

We study the effect of four compiler optimizations on the error resilience of soft computing applications. While these applications tolerate most hardware faults, they do not tolerate certain faults, that lead to large deviation in outcomes. We term such outcomes as *Egregious Data Corruptions* (EDCs). The resilience is the ability of an application to prevent the fault from becoming an EDC.

By performing fault injection experiments on the unoptimized and optimized versions of soft computing applications, we analyze how the optimizations affect the baseline error resilience (without any detection technique in place). For example, they reduce the resilience by increasing the number of EDC outcomes. We find that certain optimizations lower the baseline resilience, and this varies between benchmarks.

We also study the effect of compiler optimizations on our detector placement technique (developed as part of prior work) that places replication based detectors in code. We define an optimization to be safe under our technique, when it guarantees a certain EDC coverage compared to the unoptimized code. Based on our results on the effect of optimization on the baseline resilience and the coverage obtained by our technique, we find that to guarantee a certain error resilience for soft computing applications under compiler optimizations, efforts should be directed towards identifying *which* code regions are more susceptible to faults under these optimizations. In cases where the baseline resilience is lowered, our technique still maintains the overall resilience since the locations affected by the faults are protected by our technique. However, in cases where the baseline resilience of applications is maintained, optimizations such as SCCP and Inst-Combine make certain code regions more susceptible to faults. These locations are not protected by our technique, leading to a lower EDC coverage. As part of future work, we will study more optimizations, and group these optimizations into resilience packages, each package representing a certain level of resilience. This will help developers to choose optimizations for soft computing applications, while guaranteeing a certain degree of error resilience.

[4]Loop-Unroll does not fit in this category, as it transforms code but unrolling does not happen in most cases

Acknowledgment

We thank the anonymous reviewers, Sasa Misailovic, Sathish Gopalakrishnan and Elizeu Santos-Neto for their comments that helped improve the paper. This work was supported in part by a Discovery grant and an Engage Grant, from the National Science and Engineering Research Council (NSERC), Canada.

8. REFERENCES

[1] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. PACT, pages 72–81, 2008.

[2] M. Demertzi, M. Annavaram, and M. Hall. Analyzing the effects of compiler optimizations on application reliability. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 184 –193, nov. 2011.

[3] P. Dubey. Recognition, mining and synthesis moves computers to the era of tera. *Technology@ Intel Magazine*, pages 1–10, 2005.

[4] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. Low-cost program-level detectors for reducing silent data corruptions. DSN, pages 181–188, 2012.

[5] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. DSN, pages 135–144, 2002.

[6] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. CGO, pages 75–86, 2004.

[7] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. MICRO, pages 330–335, 1997.

[8] M. Leeke, S. Arif, A. Jhumka, and S.S. Anand. A methodology for the generation of efficient error detection mechanisms. DSN, pages 25–36, 2011.

[9] Xuanhua Li and D. Yeung. Application-level correctness and its impact on fault tolerance. HPCA, pages 181 –192, 2007.

[10] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. ICSE, pages 25–34, 2010.

[11] N. Oh, P.P. Shirvani, and E.J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability,*, 51(1):63–75, mar 2002.

[12] K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer. Application-based metrics for strategic placement of detectors. PRDC, dec. 2005.

[13] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 389–398, Washington, DC, USA, 2002. IEEE Computer Society.

[14] Vilas Sridharan and David R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *HPCA*, pages 117–128. IEEE Computer Society, 2009.

[15] D.D. Thaker, D. Franklin, J. Oliver, S. Biswas, D. Lockhart, T. Metodi, and F.T. Chong. Characterization of error-tolerant applications when protecting control data. IISWC, pages 142 –149, 2006.

[16] Anna Thomas and Karthik Pattabiraman. Error detector placement for soft computing applications. DSN, 2013.

[17] Anna Thomas and Karthik Pattabiraman. LLFI: An intermediate code level fault injector for soft computing applications. SELSE, 2013.

[18] Jeffrey D. Ullman and Alfred V. Aho. Addison-Wesley Pub. Co., Reading, Mass, USA, 1977.

[19] L.A. Zadeh. What is soft computing? *Soft computing*, 1(1):1–1, 1997.