Finding Resilience-Friendly Compiler Optimizations using Meta-Heuristic Search Techniques

Nithya Narayanamurthy, Karthik Pattabiraman, and Matei Ripeanu,
Department of Electrical and Computer Engineering,
University of British Columbia (UBC), Vancouver, Canada
{nithyan,karthikp,matei}@ece.ubc.ca

Abstract—**With the projected future increase in hardware error rates, application software needs to be resilient to hardware faults. An important factor affecting an application's error resilience and vulnerability is the set of optimizations used when compiling it. We propose an automated technique based on genetic algorithms to find the application-specific set of compiler optimizations that can boost performance without degrading the application's error resilience. We find that the resulting optimized code has significantly better error resilience than when being compiled with the standard optimization levels (i.e., O1, O2, O3), while attaining comparable performance improvements, thus leading to lower overall vulnerabilities.**

Keywords: Compiler optimizations, fault injection, genetic algorithms, resilience, vulnerability

I. Introduction

Transient hardware faults (i.e., soft errors) are becoming more frequent as feature sizes shrink and manufacturing variations increase [1]. Unlike in the past, when such faults were handled predominantly by the hardware, researchers have predicted that hardware will expose more of these faults to the software application [2], [3]. This is because the traditional methods of handling hardware faults such as dual modular redundancy and guard banding often lead to high energy overheads [4]. These faults are especially a concern for high-performance computing applications (HPC) which run for long periods of time on large scale machines.

One of the most important decisions a programmer of HPC applications must make is whether to run compiler optimizations on it. Compiler optimizations typically make programs run faster thereby boosting performance. As a result, optimizations make the program less vulnerable to hardware errors as it runs for a shorter time. However, optimizations also change the code structure of a program often removing redundancy, thereby making the program less error resilient or less capable of handling an error should it occur. Thus, the effect of compiling a program with optimizations on its overall reliability is unclear. The question we ask is: "Do compiler optimizations hurt or improve reliability?".

Prior work [5], [6] has investigated this question by studying the effect of compiling with the *standard optimization levels* (i.e., O1, O2 and O3) on programs' error resilience and vulnerability. While this is useful, the standard optimization levels group together many optimizations, and hence these papers do not disambiguate the effects of individual optimizations. Thomas et al. [7] have considered the effect of individual optimizations on error resilience, but they limit themselves to soft-computing applications, or those applications that are inherently error tolerant, e.g., multimedia applications. Similarly, Jones et al [8] have looked at the effect of individual compiler optimizations on the architectural vulnerability factor (AVF) of an application with the goal of finding optimizations that do not substantially degrade the AVF. However, the AVF does not take the final outcome of the program into account, for example, whether the fault results in a Silent Data Corruption (SDC) or a crash. For HPC applications, SDCs are often the most important failure outcome of an application as crashes can be recovered through checkpointing.

In this paper, we first perform an experimental study using fault-injection to understand the effect of individual optimizations on a program's error resilience. Although vulnerability is often the key concern in practice, it incorporates both the execution time of the program and the code structure. Resilience on the other hand, depends only on the code structure. Because optimizations typically reduce execution time we primarily focus on their effects on the code structure, i.e., resilience. We define resilience as the conditional probability that given that an error affects the program, it does not result in an SDC. We find that there is a significant difference in the error resilience achieved by individual optimizations, and that this effect varies significantly across applications. Further, *contrary to what prior studies have shown [5], [6], we find that there are compiler optimizations that can improve the error resilience of the program in addition to its performance*.

Based on the above insight, we devise an automated technique to find, for a given application, a sequence of compiler optimizations that preserves its error resilience while improving its performance, thus reducing its overall vulnerability[1]. As the space of all possible optimizations is extremely large, we leverage Genetic Algorithms (GA), a meta-heuristic search technique. GAs have been used in prior work to find compiler optimization sequences that optimize for performance [9], [10], energy consumption [11], and code size [12]. None of them consider error resilience, which is our focus. *To the best of our knowledge, we are the first to use a meta-heuristic search algorithm such as GA to find compiler optimization*

[1]In this work, we focus primarily on compiler optimizations for sequential programs, as this is what most commodity compilers currently support. Extending our results to parallel programs is a subject of future work.

sequences that can improve performance of an application without degrading its error resilience.

Our contributions are:

- Study the effect of individual optimizations on different programs' error resilience through fault-injection experiments (Section III),
- Propose a GA-based technique to find an optimization sequence for a given application that does not degrade the error resilience (Section IV),
- Implement the technique in a production, open-source compiler, LLVM [13] (Section V),
- Experimentally tune the parameters of the GA approach to achieve fast convergence to solution (Section VII) ,
- Evaluate the GA-based technique on 12 programs from the PARSEC [14] and Parboil [15] benchmark suites using fault-injection experiments, in terms of its error resilience, performance and vulnerability, and compare it to the standard optimization levels.

Our experimental evaluation (Section VII) shows that:

- The GA-based technique is able to find optimization sequences that maintain or improve application resilience compared to both the resilience of the original application and that of the application compiled using the standard optimization levels, O1, O2 and O3;
- The performance of the optimized code with our GA-based technique is on par with the performance of the code optimized with the standard optimization levels (GA based is generally better than O1, O2 and slightly worse than O3 by 0.39%);
- On average, the GA-based technique considerably *lowers* the overall vulnerability of the application (8.12 ($\pm$0.21)) compared to the unoptimized version (9.25 ($\pm$0.25)). On the other hand, the standard optimization level O1 *increases* the overall vulnerability of the application (O1-9.53 ($\pm$0.25) on average), while O2 and O3 lower it slightly (O2-9.22 ($\pm$0.24) and O3-9.11 ($\pm$0.24)). Thus, the GA-based technique significantly reduces the overall application vulnerability (by 11%) compared to the standard optimization levels and the unoptimized code.

Our results thus demonstrate that compiler optimizations need not necessarily degrade resilience, and for a very small performance loss, can lead to improved resilience and lower vulnerability than the standard optimization levels (and even the original program).

II. Background and Fault Model

In this section, we first define our metrics, error resilience and vulnerability. We then present a brief overview of genetic algorithms, and describe our fault model.

A. Error Resilience and Vulnerability

A hardware fault can cause a program to fail in one of three ways: it may cause the program to crash, hang, or have a silent data corruption (SDC). SDC is an outcome that results in incorrect output without any indication, hence the name "silent". We focus on SDCs as they are considered the most severe kind of failures in a program (the other failures, namely crashes and hangs, can be detected through hardware exceptions and timeout mechanisms respectively).

Error resilience is the ability of a program to prevent an error that has occurred during runtime from becoming an SDC. In other words, resilience is the conditional probability that a program does not produce an SDC given that it is affected by a hardware fault (i.e., the fault is activated). Resilience is a characteristic of the application's code structure, and is independent of its execution time. Vulnerability on the other hand, takes execution time into account to account for the higher likelihood of a fault for a longer running program.

More precisely, we define the $Resilience = (1 - SDCrate)$, and $Vulnerability = (SDCrate * Executiontime)$, where $SDCrate$ is the fraction of SDCs observed over the set of all activated faults (i.e., faults that manifest to the software).

Note that our definition of vulnerability differs from the commonly used notion of the Architectural Vulnerability Factor [16], which is defined in terms of the number of bits in a hardware structure that are needed for architecturally correct execution (ACE). We eschew this definition as it is tied to the architectural state of the processor, while we want to capture the effect of the error on the application. Further, AVF studies often employ detailed micro-architectural simulators which are slow, and hence do not execute the application to completion. On the other hand, we want to execute applications to completion on the real hardware as we are interested in the ultimate effect of the error (i.e., whether or not it results in an SDC).

As mentioned earlier, we focus on resilience to separate the effects of compiler optimizations on code structure and execution time. Since all the optimizations we choose aim at improving performance, the vulnerability will be reduced if the error resilience is maintained the same after the optimization is applied (due to shorter execution time). We show later in the paper that the convergence of our approach when choosing optimizations for resilience is much faster than directly choosing them for vulnerability, while yielding comparable results (Section VII-D).

B. Genetic Algorithm (GA)

A Genetic Algorithm (*GA*) [17] is a meta-heuristic search algorithm that is inspired by natural evolution. The algorithm starts with an initial set of candidate solutions. They are collectively called as the *Population*. The algorithm has a fitness function that is used to calculate a candidate's *fitness score*. The fitness score depends on how good the candidate is at solving a problem, and it is the parameter that evaluates a candidate's rank towards the optimal solution. One or two candidates are chosen from the population to perform *recombination* at each stage.

The recombination operations are of two types: *Crossover* and *Mutation*. Two candidates undergo Crossover whereas, for mutation, only one candidate takes part. The crossover operation performs a randomized exchange between solutions, with the possibility to generate a better solution from a good one. This operation tends to narrow the search and move towards a solution. On the other hand, mutation involves flipping a bit or an entity in a solution, which expands the search exploration of the algorithm. Crossover and mutation rate are the probabilities at which the respective operations are performed [18] [19]. The choice of these probability values reflects the trade-off between exploration and exploitation (or convergence). A higher mutation rate for example, leads to better exploration but can delay convergence. On the other hand, a high crossover rate can lead to faster convergence, but may get stuck in a local maxima.

Typically, recombination gives rise to new better performing members, which are added to the population. Members in the population that have poor fitness scores are thus eliminated gradually. This process is repeated iteratively until either a population member has the desired fitness score, thereby finding a solution, or the algorithm exceeds the time allocated to it and is terminated.

C. Fault Model

In this paper, we consider transient hardware faults that occur in the computational elements and register file of the processor. These faults occur when particle strikes or cosmic rays affect the flip-flops or the logic elements. Particle strike or cosmic rays might impact various chip components, namely memory, instruction cache, data cache, ALU, pipeline stages. Memory and cache are typically protected by error correcting codes or parity. They have the ability to correct/detect single bit flips caused by the particle strike. Therefore, we do not consider faults that affect memory. Likewise, faults occurring in the instructions' encoding can be detected by the use of simple codes - therefore we do not consider these faults either. However, when a particle strikes the computational components like the ALU, registers, processor pipelines, logic gates etc, they affect the result of the instruction that is currently being executed in that component. This faulty result is consumed by the subsequent dependent instructions ultimately impacting the application's outcome if allowed to propagate.

Cho et. al. [20] have found that there may be significant differences in the raw rates of faults between fault injections performed in the hardware and software. However, we are interested in faults that are not masked by the hardware and make their way to the software application. Therefore, we inject faults directly at the application level. A similar fault model has been used by prior work in this area [3], [2], [21].

III. Initial Study

In this section, we perform an initial fault-injection study that analyzes the effect of individual compiler optimizations on error resilience. The experimental setup and the benchmarks considered here are described later in Section VI.

Fault Injection Results: We chose 10 individual optimizations at random from about 50 optimizations available in the LLVM compiler [13]. We performed an initial study to analyze

TABLE I: Different optimizations used in the initial study

Optimization	Expansion
licm	Loop Invariant Code Motion
inst-combine	Instruction Combine
cse	Common Subexpression Elimination
gvn	Global Value Numbering
ip-sccp	Inter-procedural Sparse Conditional Constant Propagation
inline	Function Inlining
loop-reduce	Loop Operator Strength Reduction
loop-unroll	Loop Unrolling
loop-unswitch	Loop Unswitching
sccp	Sparse Conditional Constant Propagation

the impact of individual optimizations on the error resilience of two applications from the PARSEC benchmark suite, namely *Blackscholes* and *Swaptions*. We first compiled the programs with each of the ten chosen optimizations using LLVM. The optimizations chosen are shown in Table I.

We performed fault injection experiments on the unoptimized version and the ten different optimized versions of the programs to measure their respective error resilience values. We performed a total of 3000 fault injection runs and computed error bars at the 95% confidence intervals. Figure 1 shows the resilience (in %) of the different versions of the two programs compared to the resilience of the unoptimized version (baseline). The figure shows that some optimizations degrade the error resilience of the program, while some optimizations improve the resilience. For example, the *loop-reduce* optimization improves the error resilience of Blackscholes, while *instcombine* degrades the error resilience. Further, the resilience effect of an optimization differs from one application to another. For example, while the *loop-reduce* optimization improves the resilience of *Blackscholes*, it degrades that of *Swaptions*.

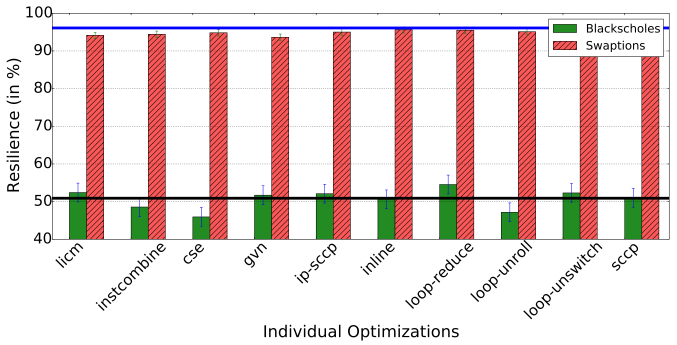

Fig. 1: Resilience of blackscholes and swaptions optimized with different individual optimizations (Black line at bottom represents the resilience of the unoptimized version of Blackscholes; Blue line at top represents the resilience of the unoptimized version of Swaptions). Error bars are for the 95% confidence interval.

To further understand why individual optimizations enhance or degrade a program's error resilience, we wrote a series of micro-benchmarks that each attempt to exercise a single optimization. We then performed fault-injection studies into these micro-benchmarks in order to study the effect of these optimizations. This gives us an idea of why a particular optimization increases or decreases error resilience. We give two examples, one of an optimization that degrades resilience, and the other of an optimization that enhances resilience.

Resilience degrading optimization: Consider the commonly used loop optimization *loop-invariant code motion (LICM)*, which attempts to reduce the operations performed inside loops. It moves the loop-invariant expressions inside the loop to the pre-header block of the loop without affecting the semantics of the program.

Figure 2a shows a code snippet (unoptimized) from the micro-benchmark, and Figure 2b shows the code optimized by the LICM optimization. Our original code snippet includes multiple such loops with similar operations - however, we show only one loop for simplicity. It can be seen that the expression that computes *alpha* (line 3 in Figure 2a) inside the loop does not depend on the induction variable of the loop. Thus the LICM optimization moves those expressions to the pre-header block of the loop and minimizes the computations performed inside the loop as shown in Figure 2b.

```
1 for(i=0; i<10; i++)
2 {
3   alpha=(x*c)+s;
4   rs1[i]=
          i+(alpha*7);
5 }
```

```
1 alpha=(x*c)+s;
2 for(i=0; i<10; i++)
3 {
4   rs1[i]=
          i+(alpha*7);
5 }
```

Fig. 2: Effect of running the LICM optimization on a code snippet (a) Unoptimized version, (b) Optimized version.

From our fault injection experiments, we observed that the LICM optimization reduces the error resilience of the program compared to the unoptimized version. To understand why the resilience is degraded, assume that the LICM optimized code experiences a fault in the computation `alpha = (x * c) + s` (line 1 in Figure 2b). This fault will affect all values of the array `rs1` in all loop iterations. The original code on the other hand, computes `alpha = (x * c) + s` (line 3 in Figure 2a) on every iteration of the loop, and hence a fault in the computation affects only the values of the array in that loop iteration, namely `rs1`. Therefore, the LICM transformed code has a greater likelihood of experiencing an SDC due to the fault, and its resilience is lowered. This is an example of how an optimization may lower the error resilience of an application.

Resilience enhancing optimization: Consider another loop optimization *loop strength reduction (LOOP-REDUCE)*, that performs strength reduction on array references by replacing complex operations inside the loop involving the loop induction variable with equivalent temporary variables and simpler operations. Similar to the previous example, Figure 3a shows a sample code snippet and how it is transformed by the LOOP-REDUCE optimization. The loop induction variable that is used for array references and value computation in the expression, `rs1[i] = i*alpha` (line 4 in Figure 3a) is replaced with temporary variables `temp` and `temp1` for the address and value of array `rs1` as shown in Figure 3b (line 6-8). Hence the induction variable here is only used to control the loop entry and exit after the optimization.

From our fault-injection experiments, we observed that the LOOP-REDUCE optimization enhances the resilience of the program compared to the unoptimized version. To understand why the resilience is enhanced, consider a fault that occurs in the computation of the loop induction variable. In the unoptimized version, the fault would affect the value and references of array `rs1`. On the other hand, in the optimized version, the loop induction variable is restricted to the role of iterating and exiting the loop, and a fault occurring in this induction variable would not affect the array reference and its contents. Thus the optimized version is more resilient that the unoptimized version. This example shows how an optimization can improve the error resilience of an application.

```
1 alpha=(x*c)*s;
2 for(i=0; i<10;i++)
3 {
4   rs1[i]=i*alpha;
5 }
```

```
1 alpha=(x*c)*s;
2 temp=&rs1;
3 temp1=0;
4 for(i=0; i<10;i++)
5 {
6   *temp=temp1*alpha;
7   temp1=temp1+1;
8   temp=temp+
          sizeof(int);
9 }
```

Fig. 3: Effect of running the LOOP-REDUCE optimization on a code snippet (a) Unoptimized version, (b) Optimized version.

Therefore, different optimizations have different effects on a program's error resilience, with some optimizations degrading resilience and others improving it. Further, it is often difficult to judge apriori whether an optimization will lower or improve the error resilience, as it is dependent on the application's characteristics. This is why we need an automated method to find optimization sequences for an application that preserve its error resilience with respect to the unoptimized version.

IV. Methodology

In this section, we first present the problem statement and discuss its complexity. We then present our GA-based approach for solving the above problem.

A. Problem Statement and Complexity

We devise an automated method to solve the following problem: given a program P, find an optimization sequence that provides performance improvement without degrading resilience. If $\gamma = [\alpha_1, \alpha_2, \alpha_3, ...\alpha_n]$ where $\alpha_1, \alpha_2, \alpha_3, ..\alpha_n$ are individual compiler optimizations and γ is the superset of optimizations, our goal is to find a non-empty optimization sequence $\varphi = \{\alpha_{x1}\alpha_{x2}...\alpha_{xt}\}$, where $1 \leq x1, x2, ..xt \leq n$, that retains the resilience of the program, i.e., $Resilience(\varphi(P)) \geq Resilience(P)$ and $|\varphi| \geq 1$. The latter constraint is necessary to prevent the trivial solution where φ

is an empty set, i.e., when no optimizations are performed on the program and the resilience is the same.

Note that a modern compiler has more than 50 optimizations at its disposal. So a naive search strategy to solve this problem would have to search through 2^{50} combinations, simply to find the sets of optimizations to run on the program. Each set can in turn be permuted in different ways (with repetitions allowed), and hence there is an exponential number of possibilities for solving this problem. Also, as mentioned in our initial study in Section III, the effect of optimizations varies significantly across applications. Hence we need an efficient way to search the space of optimizations for resilience, which a meta-heuristic search method such as our GA-based method provides. While other meta-heuristic search methods are also possible (e.g., Simulated annealing), we use GA as we found that they yield high quality solutions in a small period of time.

B. GA-Based Approach

We explain our GA-based approach for finding the appropriate compiler optimization sequence for an application that does not degrade its error resilience. We begin with a set of unique individual compiler optimizations as our initial population. In GA terms, these individual optimizations constitute the gene and the resulting combinations of optimizations constitute the chromosomes. The optimizations can consist of all the optimizations available in a standard optimizing compiler such as *gcc* or *llvm*. We obtain the initial error resilience of the unoptimized version of the application through fault injection experiments. This is the target error resilience for the algorithm.

The GA-based algorithm is presented in Algorithm 1. The steps are further explained as follows.

1. *Initialization:* Every individual member of the population is called as a candidate. The candidates in the initial population are unique individual compiler optimizations. The fitness score of every candidate in the population is calculated using the fitness function (discussed in Step 2). This is shown in the initialization part of the Algorithm 1. The size of the initial population determines the convergence rate of the algorithm and the quality of its solution. We experimentally choose the initial population size in Section VII.

We first check if there is any candidate in the initial population that does not lower the program's error resilience. If such a candidate exists, then it is considered as an optimal candidate solution with the desired resilience and the algorithm terminates (lines 2-4). This is a trivial condition and is unlikely to occur. For example, we did not encounter this condition in any of our experiments.

2. *Fitness Function:* In GA, the fitness score of a candidate is used to determine whether the candidate should be carried forward to the next generation. We need a fitness function($\Theta()$) that measures the error resilience of a candidate optimization sequence. The fitness function can be based on a resilience model or on fault injection experiments. We use fault injection for this purpose. However, it is also possible to use models for

Algorithm 1 Algorithm 1: GA-based approach to find an optimization sequence that does not degrade error resilience

```
$\alpha_1, \alpha_2, \alpha_3, ... \leftarrow$ Individual optimizations
$\Theta() \leftarrow FitnessFunction()$
$s_{min} \leftarrow$ Minimum fitness score of population
$\alpha_{min} \leftarrow$ Candidate with fitness score $s_{min}$
$s_{max} \leftarrow$ Maximum fitness score of population
$\alpha_{max} \leftarrow$ Candidate with fitness score $s_{max}$
$s_{target} \leftarrow$ Resilience of unoptimized version
$\delta_c \leftarrow CrossoverRate$
$\delta_m \leftarrow MutateRate$
population $\leftarrow [(\alpha_1, \Theta(\alpha_1)), (\alpha_2, \Theta(\alpha_2)), (\alpha_3, \Theta(\alpha_3)), ...]$
Input: Source code, population
Output: Optimization sequence that retains the resilience of the given source code
 1: procedure OPTIMIZATION SEQUENCE FOR RESILIENCE
 2:     $s_{max} = max(\Theta(\alpha_1), \Theta(\alpha_2), \Theta(\alpha_3), ...)$
 3:     $\alpha_{max} = getCandidate(population[s_{max}])$
 4:     while $s_{max} \leq s_{target}$ do
 5:         $\alpha_a, \alpha_b = TournamentSelection(population)$
 6:         if $Random() < \delta_c$ then
 7:             $\hat{\alpha} = crossover(\alpha_a, \alpha_b)$
 8:         else
 9:             $\hat{\alpha} = \alpha_a$
10:         end if
11:         if $Random() < \delta_m$ then
12:             $\hat{\alpha} = mutation(\hat{\alpha})$
13:         end if
14:         $s_{min} = min(\Theta(\alpha_1), \Theta(\alpha_2), \Theta(\alpha_3), ...)$
15:         if $s_{min} < \Theta(\hat{\alpha})$ then
16:             $\alpha_{min} = getCandidate(population[s_{min}])$
17:             $Eliminate(population, \alpha_{min})$
18:             $Add(population, (\hat{\alpha}, \Theta(\hat{\alpha})))$
19:         end if
20:         $s_{max} = max(\Theta(\alpha_1), \Theta(\alpha_2), \Theta(\alpha_3), ...)$
21:         $\alpha_{max} = getCandidate(population[s_{max}])$
22:     end while
23: return $\alpha_{max}$
24: end procedure
```

predicting the resilience of an application based on its code structure that have been proposed in recent work [21], [22].

The fitness function Θ is used to rank the resilience of the candidate. Based on this rank, the GA decides whether the candidate should be considered for the next evolution round. It is important to ensure that we can obtain tight confidence intervals on the error resilience as we use it to compare solutions with each other in terms of resilience. Therefore, we perform a few thousand fault injection experiments in each iteration of the GA to determine the fitness score, depending on the benchmark's characteristics.

3. *Tournament Selection:* The goal of the tournament selection is to determine which pair of candidates should be recombined with each other to form the next generation of

the GA. In our algorithm, we choose two candidates from the population based on a heuristic(line 5). We consider two different heuristics: (i) Random selection (ii) Score based selection. In random selection, we pick two candidates randomly from the population. In score based selection, we pick the two best candidates (top two fitness scores), the intuition being that the fittest candidates can give rise to better offspring. We evaluate the effectiveness of both these heuristics in Section VII.

4. *GA Recombination operations:* We perform recombination operations on the candidates chosen from tournament selection (line 6 -13). Recombination operations are of two types: (i) Crossover (ii) Mutation. *CrossoverRate* and *MutateRate* determine the probability at which these operations are performed. The *CrossoverRate* is chosen as suggested and used by classical papers in the GA area [23], [24], [25], [26], [27]. The *MutateRate* was chosen based on our analysis discussed in the Section VII, as there is no consensus in the literature on this value.

We devise new crossover and mutation operations in order to explore the large space of optimizations and drive the algorithm towards obtaining an optimization set that retains the resilience. We briefly describe these operators.

(i) *Crossover:* Crossover operation involves either append or swap operations. These operations increase the chances of combining the sequences of the two chosen candidates to evolve a new candidate with a higher resilience. The append operation simply appends the entities of the two selected candidates The swap operation is similar to the two-point crossover, where the entities within the two selected index are swapped between the candidates.

(ii) *Mutation:* In some cases, we found that a single compiler optimization in the candidate set degrades the overall resilience, and hence by replacing it with another individual optimization or deleting it, the GA can generate a better candidate. Thus we devise a mutation operation to add, delete or replace an individual optimization with another one.

5. *Elimination:* The goal of the elimination step is to eliminate the unfit candidates from the population. The fitness score of the weakest candidate from the population is compared with the fitness score of the new candidate generated from the recombination operations. If the weakest candidate's fitness score is smaller than that of the new candidate, it is eliminated from the population. In this case, the new candidate with a better resilience is added to the population, hence will be considered for the next generation's evolution. If its fitness score is not smaller, the new candidate is not added to the population, and the population remains unmodified. This is shown in lines 14-19 in Algorithm 1. The main intuition here is that weaker candidates have lower probability of giving rise to stronger offspring, and hence need to be eliminated from the pool of candidates to carry forward to the next generation.

6. *Termination:* If a new candidate is added to the population, we check whether its resilience is greater than or equal to the target resilience i.e., the resilience of the unoptimized program. If this is the case, we call it the *candidate solution* and stop the algorithm (line 4, and line 23). Otherwise, we repeat the above steps of *Recombination* and *Elimination* until we obtain a candidate solution. It is possible that such a candidate solution takes too long to obtain, or is never obtained. To resolve this, we terminate the algorithm if the average fitness score of the entire population does not change for numerous generations. In this case, the algorithm returns the best candidate from the population that is closest to the resilience of the unoptimized version as the candidate solution.

V. Implementation

We implemented our approach using the LLVM compiler [13] and the open-source LLFI fault injection tool [28]. LLVM is a popular optimizing compiler that includes a host of standard optimizations. We leverage the machine-independent optimizations in the LLVM compiler, and do not consider backend optimizations for portability. For seeding our technique, we pick a subset of optimizations consisting of dataflow, loop, global and a few other optimizations available in the LLVM compiler [13]. This subset comprises around 10 different optimizations (we explain why 10 in Section VII).

For the fitness function in our approach, we use LLFI, a fault injection tool that operates at the LLVM Intermediate Representation (IR) code level [13] to inject hardware faults into the program's code. We use LLFI as it has been found to be accurate for measuring the SDC rate of an application relative to assembly-level fault injection [28].

LLFI first takes the IR code as the input and determines the target instructions/operands for fault injection. It then instruments the target instructions/operands with appropriate calls to the fault injection functions. These fault injection functions are the ones that inject faults. In each fault injection run, the compiled program is executed, and LLFI randomly chooses a single dynamic instance of the instrumented instructions to inject a fault into. It then flips a single bit in the destination register of the instruction (after it is written). We consider single bit flips as it is the de-facto model for simulating transient faults. Note that the optimizations are run before the instrumentation by LLFI to prevent the instrumentation from interfering with the optimizations.

VI. Experimental Setup

Benchmarks: We evaluate our technique on twelve programs, five from the PARSEC [14] and seven from the Parboil [15] benchmark suites. The benchmarks represent a wide variety of tasks ranging from video processing to scientific computing, and are all written in C/C++. They range in size from a few hundred to a few thousand lines of code. We choose these programs as they were compatible with our experimental setup (i.e., we could compile them with LLFI). The benchmarks chosen and their characteristics are shown in Table II. We use the small inputs that come with these benchmarks for our experiments, as we need to perform thousands of runs of each program for fault injections.

Tuning of the GA parameters: We first evaluate the performance of the GA approach to tune its parameters to obtain faster convergence. One way to measure performance of

TABLE II: Benchmark programs that are used in our experiments

Program	Benchmark suite	Description
Blackscholes	PARSEC	Computes the price of options using blackscholes partial differential equation.
Swaptions	PARSEC	Computes the price of portfolio of swaptions by employing Monte Carlo(MC) Simulations.
x264	PARSEC	An H.264/AVC video encoder, that achieves higher output quality with lower bit rate.
Fluidanimate	PARSEC	Simulates an incompressible fluid for interactive animation purposes.
Canneal	PARSEC	Minimizes the routing cost of a chip design using a cache-aware simulated annealing.
Bfs	Parboil	Implements a breadth first search algorithm that computes the path cost from a node to every other reachable node.
Histo	Parboil	Computes a 2-D saturating histogram with a maximum bin count of 255.
Stencil	Parboil	An iterative Jacobi solver of heat equation using 3-D grids.
Spmv	Parboil	Implements a Sparse-Matrix Dense-Vector Product
Cutcp	Parboil	Computes short-range electrostatic potentials induced by point charges in a 3D volume
Sad	Parboil	Computes sum of absolute differences for pairs of blocks which is based on the full-pixel motion estimation algorithm
Sgemm	Parboil	Performs a register-tiled matrix-matrix multiplication

the algorithm is by using wall clock execution time. However, the execution time for the GA is dominated by the time it takes to perform the fault injections in each iteration of the GA to evaluate the fitness of each candidate[2]. Therefore, the number of generations taken by the algorithm is a more meaningful measure of performance, as the greater the number of generations, the more the number of candidate sequences generated, and hence the more the number of total fault injections that must be performed to evaluate the candidates. We consider the effects of the following parameters. These parameters are explained in Section IV.

(1) *MutateRate*: We vary this value based on what the literature on GA recommends [29], from low to high values.

(2) *Population size*: We vary this value from 10 to 40 as we have a total of 50 optimizations in LLVM.

(3) *Tournament selection strategy*: We consider two strategies, *random selection* and *score-based selection*.

(4) *Optimization Types*: We vary the optimization types (data-flow, loop, global and others) in the population.

Once we tune the GA parameters, we use these values to derive the candidate solutions used in the resilience and performance evaluation experiments described next.

Resilience Evaluation: We first compile each of the programs using LLVM with the -O0 option (no optimizations) for generating the unoptimized program. We then measure its resilience by performing fault injection experiments using the fault injection tool LLFI (as explained in Section V). We consider this as the baseline for our experiments. We perform a total of 1000 fault injection experiments per benchmark program (one fault per run). The error bars range from 0.85% to 2.501% depending on the benchmark's SDC rate, for the 95% confidence interval.

We compare the results of our technique with the standard optimization levels O1, O2 and O3, as this is what prior work has used for evaluating error resilience of compiler optimizations [5], [6]. We repeat the above process for each of the optimization levels O1, O2, and O3, for each benchmark program, and obtain their error resilience values. We then run our GA-based optimization to identify a candidate solution (i.e., optimization sequence) for each benchmark program, and then repeat the same experiment for this candidate solution. We compare each of the resilience values to the baseline resilience of the unoptimized version.

There are a total of 60000 injections performed in our experiments (12 benchmarks, 1000 injections, 5 executables namely, O1, O2, O3, unoptimized and candidate solution).

Performance Evaluation: We measure the execution time of the executable compiled with the appropriate set of optimizations i.e., O1, O2, O3 and the candidate solution on our hardware platform. The platform we use is a Intel E5 Xeon machine with 32G memory running Ubuntu Linux 12.04. We measure the execution time by taking an average of 10 trials (standard deviation ranging from 0.0031 to 0.028). The average value of the error bars is 1.71% for the 95% confidence intervals.

VII. Results

We first present the results of how we tune the parameters of our GA-based algorithm for faster convergence. We then present the results for evaluating the error resilience of the candidate solution (i.e., optimization sequences) found by our GA-based approach, and that of the standard optimization levels. We also present the results of the performance improvement and vulnerability reduction of each of these optimization sequences compared with the unoptimized version. Finally, we compare our resilience-enhancing fitness function with one that optimizes directly for the vulnerability, with an unbounded GA-based approach (GA-based approach without a predefined conditional termination), and with a random walk based technique.

A. Effect of GA Parameters

We consider the effect of four parameters on our GA-based approach's convergence rate. Due to space constraints, we only present the results for two benchmarks, *bfs* and *blackscholes*, but we obtained similar results for the other programs. We do not consider the quality of the solution obtained in this experiment - rather, our goal is only to tune the algorithm for fast convergence.

Mutation Rate: We first considered the effect of varying the mutation rate of the algorithm, keeping the other parameters

[2]Note however that fault injection is an instance of an embarrassingly paralllel problem, and can be accelerated using large clusters or clouds.

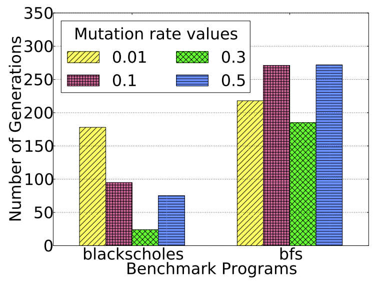

Fig. 4: Number of generations taken to generate the candidate solution with different mutation rate values.

fixed. As explained tion II, this parameter represents the trade-off between search space exploration (leading to potentially better solutions) and convergence (leading to faster solutions). A larger mutation rate is associated with better exploration but slower convergence.

Figure 4 shows the effect of varying the mutation rate on the convergence rate. We performed these experiments with four different mutation rate values: 0.01, 0.1, 0.3, 0.5. We observe that the algorithm that obtains the candidate solution in the least number of generations is for mutation rate = 0.3. *Therefore, we choose a mutation rate of 0.3 for our technique.*

Selection Strategy: As mentioned in Section IV-B, there are two possible selection strategies in each iteration of our algorithm. One strategy is to randomly choose any two candidates in the population to move forward to the next generation (*random*). Another strategy is to choose the two best candidates, i.e., the candidates with the highest fitness scores in each generation (*score-based*). We compared the number of generations taken by each strategy to attain convergence across the benchmark programs (all other values are kept the same). The results of this comparison are shown in Figure 5. It can be observed that for both benchmarks, the score based selection method takes more generations than the random selection to obtain the candidate solution. The poor performance of score based selection is because it moves faster, but gets stuck at local maxima, trying to select the best candidates in every generation, following which it takes a long time to attain convergence. On the other hand, the random selection method moves towards the candidate solution slower, but does not get stuck in the local maxima, making it converge faster. *We therefore use the random selection strategy in our approach.*

Population Size: This represents the number of individual optimizations present in the initial population considered by our GA-based approach. To evaluate the effect of the population size, we examined the number of generations that the algorithm takes to converge for different population sizes ranging from 10 to 40 (recall that we have around 50 optimizations).

Figure 6 shows the results of this experiment. The figure shows that the number of generations taken to attain convergence increases with the increasing population size. Hence, a smaller population size would arrive at a candidate solution faster. On the other hand, increasing the population size may

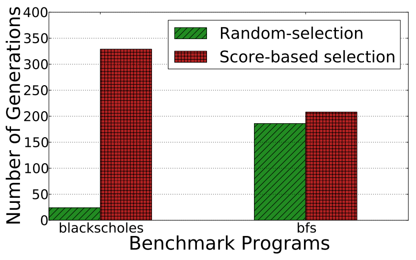

Fig. 5: Number of generations taken to generate the candidate solution by the random selection and score based selection strategies.

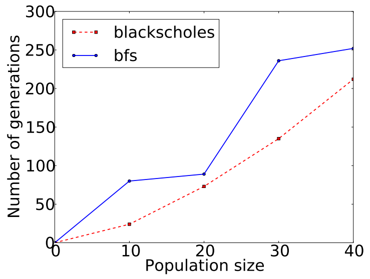

Fig. 6: Number of generations taken to generate the candidate solution with different population sizes.

lead us to a better solution. We however find that even by restricting the population size to just 10 optimizations, we are able to achieve satisfactory performance without degrading the error resilience (Section VII-C). *Based on our results, we choose a population size of 10 for our GA-based approach.*

Optimization Types: Compiler optimizations are classified into different types based on the transformation they perform on the program. They are classified into *data-flow optimizations*, *loop optimizations*, *global optimization* and *others*. For the initial population in our approach, we pick a subset that contains a combination of optimizations from the available classes. We wanted to investigate if we could achieve faster convergence by using only a specific class of optimizations as the population. For this experiment, we restrict the types of optimizations to each of the above categories, and compare the number of generations taken to obtain the candidate optimization sequence. We also compare it to the convergence rate obtained when all the categories are combined together, called "combination of all'.

The results are shown in Figure 7. From the figure, it is evident that no single class of optimization outperforms the rest for both benchmarks. This suggests that there is no one universal set of optimizations that can accelerate the convergence. Hence, we chose a *population that consists of a combination of all the optimization types in our experiments*, i.e., we do not restrict ourselves to a specific optimization type.

B. Resilience Evaluation

We first present the aggregate results across benchmarks for the unoptimized, original versions of the programs. The

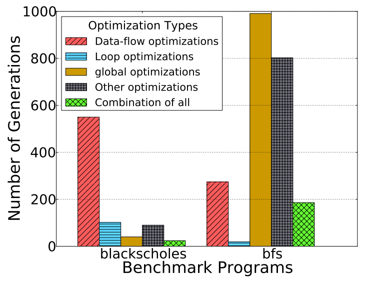

Fig. 7: Number of generations taken to generate the candidate solution with different optimization types.

average percentages of SDCs observed across all benchmarks is 19.0%. Crashes constitute 36.8% and benign injections constitute 44.2% of the injections on average (these are injections the resulted in the program completing with the correct output). We observed that hangs are negligible in our experiments. Note that we include only the activated faults in the above results, or those faults that are actually read by the system and affect the program's data - this is in line with the definition of resilience (see Section II-A).

We compare the resilience of the program compiled with the optimization sequence (i.e., candidate solution), obtained from our technique, with the resilience of the unoptimized program, and that of the compiler optimization levels O1, O2 and O3. Figure 8 shows the resilience (in %) of the unoptimized and the different optimized versions of the program (candidate solution, O1, O2 and O3). In this graph and the next few graphs, we show the error bars at the 95% confidence interval.

As can be seen in the figure, the optimization levels O1, O2 and O3 have degraded the resilience of the application compared to the unoptimized version, for all the benchmarks. *On the other hand, the candidate solution generated by our technique provides a resilience better than or on par with the unoptimized version of the program for all benchmarks.* The arithmetic mean of the error resilience across benchmarks in percentages, of the unoptimized version, GA-candidate solution, O1, O2 and O3 levels are respectively 76.14 (±1.54), 79.125 (±1.57), 72.77 (±1.6), 73.15 (±1.6) and 73.38 (±1.54) [3]. *Thus, the candidate solution achieves better average resilience than even the most aggressive optimization level, O3.*

Figure 9 shows the difference in the SDC rates of the candidate solution and the optimization levels normalized to the SDC rate of the unoptimized code. As can be seen in the figure, the SDC rates of the candidate solutions are lower by 5 to 45%, while the SDC rates of the optimization levels are higher by 5-120% depending on the program. *On average, the SDC rate of the benchmarks compiled with the candidate solution is lower than that of the unoptimized code by 20%, while the SDC rates of the benchmarks compiled with the optimization levels is higher by about 20%.* [4]

[3] ± refers to the error bars.

[4] We use the geometric means (GMs) as we are comparing the normalized SDC rates.

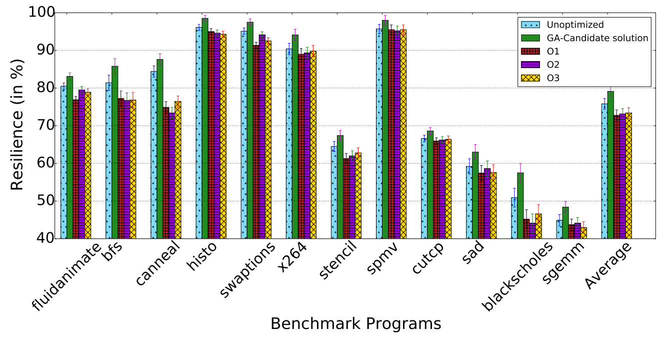

Fig. 8: Resilience of the Unoptimized, candidate solution, O1, O2 and O3 levels. (Higher values are better).

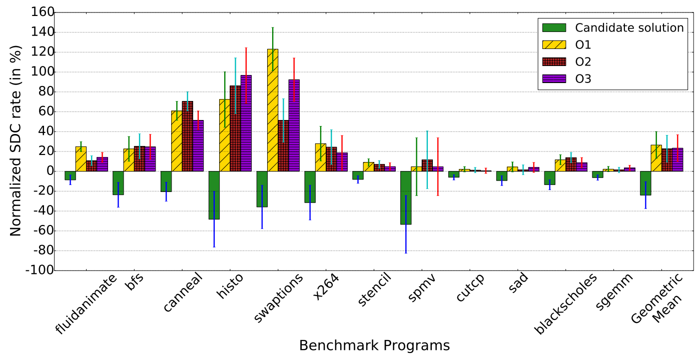

Fig. 9: SDC rate of the candidate solution, O1, O2 and O3 levels normalized to those of the unoptimized code. (Lower values are better).

C. Performance Evaluation

As in the resilience experiment, we compare the performance of the unoptimized version of the program with the candidate solution, O1, O2 and O3. Figure 10 shows the execution time (measured in seconds) of the unoptimized code, candidate solution found by our technique and the optimization levels O1, O2 and O3.

The figure shows that the candidate solution from our technique provides better performance than the unoptimized version, as do the optimization levels (as is expected). Further, the candidate solution's average of execution time across benchmarks is better than those of the optimization levels O1 and O2, and only slightly worse than O3 (by 0.39% on average). *This shows that resilience friendly optimizations obtain performance improvements that are comparable to those obtained by the standard optimization levels.*

In fact, we observe that the performance of the candidate solution found by our approach is better than the performance provided by the optimization levels in some benchmarks. For example, in the case of the *blackscholes* program, our optimization sequence provides better performance than the optimization levels O1 and O2. However, this is not the case for other benchmarks such as *fluidanimate* and *canneal*, where the candidate solution's performance is much worse than the optimization levels. Analyzing further, we observed

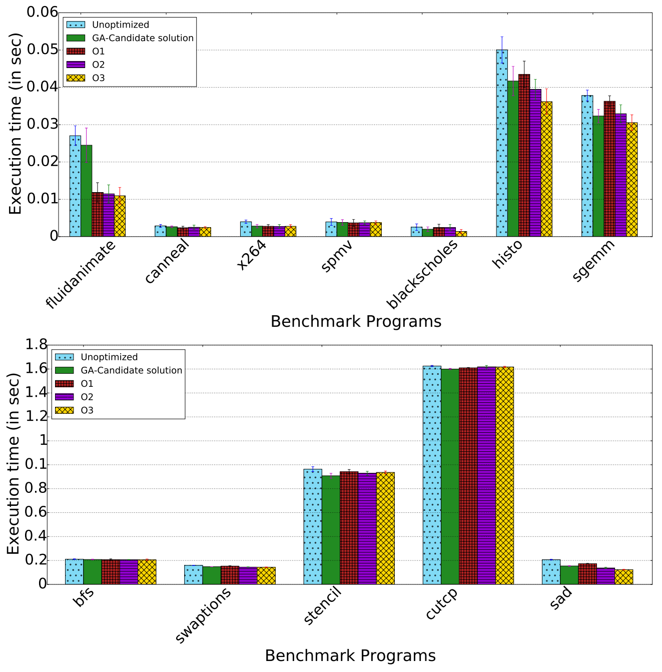

Fig. 10: Runtime of the unoptimized code, candidate solution, O1, O2 and O3 levels. (Lower values are better)

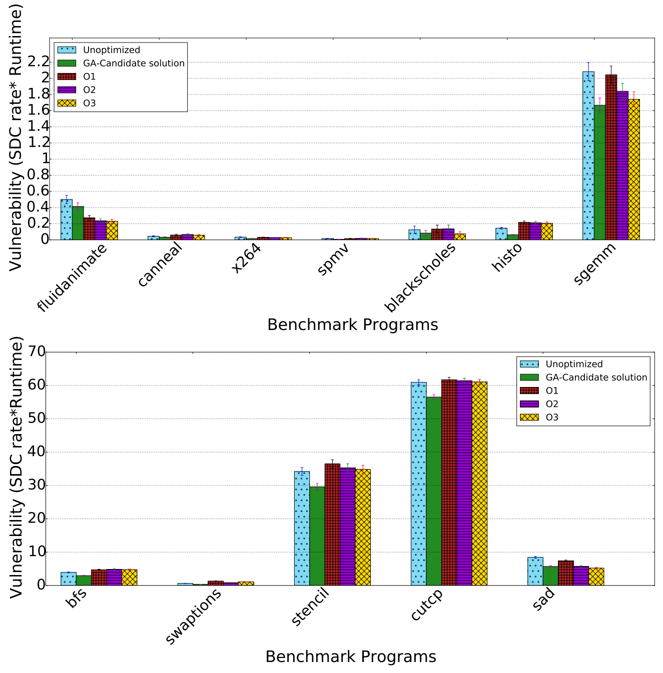

Fig. 11: Vulnerability of the unoptimized code, candidate solution, O1, O2 and O3 levels. (Lower values are better).

that the "union" data type in the *fluidanimate* program has been optimized by *-scalarrepl* optimization, and caused this major performance improvement. However, *-scalarrepl* was not included in our initial population.

D. Vulnerability Evaluation

Similar to the above evaluations, we compare the vulnerability of different versions of the program. Figure 11 compares the vulnerability of the candidate solution with the optimization levels O1, O2 and O3. The 95% confidence interval error bars for vulnerability (product of SDC rate and execution time) are calculated by adding the relative errors, which is the standard way for calculating the uncertainity of a product [30]. The figure shows that in most of the benchmark programs, the candidate solution obtained from our technique reduces the overall vulnerability of the program. On the other hand, the optimization level O1 increases the overall vulnerability of the program, while O2 and O3 both reduce the vulnerability, but not to the level of the candidate solution.

On an average, the vulnerability of programs compiled with standard optimization levels O1,O2 and O3 are 9.53 (±0.25) , 9.22 (±0.24) and 9.11 (±0.24). These are slightly higher or only marginally lower than the vulnerability of the unoptimized version (9.25 (±0.25). In comparison, the candidate solution found by our GA-based approach lowers the vulnerability by a significant amount to 8.12 (±0.21). This constitutes a vulnerability reduction of about 11%, on average, compared to even the most aggressive optimization level O3.

However, there are two exceptions to the vulnerability reduction achieved by the candidate solution, namely *fluidanimate*, and the O3 level in *sad* and *blackscholes*. In the case of *fluidanimate*, the standard optimizations reduce vulnerability much more than our candidate solution. This is because the standard optimizations reduce the program's execution time by 50% or more, which far outweighs the increased SDC rate due to the optimizations. We have explained the reason for this substantial reduction in execution time in Section VII-C.

In the case of *blackscholes* and *sad*, our candidate solutions do worse than the optimization level O3 in terms of vulnerability reduction. Again, the optimization level O3 reduces the execution time by nearly 40% in these programs, and this reduction outweighs the increase in the SDC rate, resulting in lower vulnerability. However, the difference in the vulnerability reduction between the candidate solution and the O3 optimization is small in these programs.

E. Optimizing for Vulnerability

To test if optimizing directly for vulnerability would yield better results than optimizing for resilience as we have done so far, we modified the fitness function in Algorithm 1 to optimize for the vulnerability, In this case, our goal was to find an optimization sequence that does not increase the vulnerability of the program compared to the unoptimized version. The results were similar to the above results, and hence we do not report them. However, the algorithm with the vulnerability measuring fitness function took about $1.2x$ to $10x$ the number of iterations as the original algorithm. This is because the algorithm with vulnerability measuring fitness function tries to optimize for both time and resilience, and hence has a larger state space to deal with. *This is why we optimize for resilience rather than directly for vulnerability in this paper.*

F. Resilience-Enhancing Compiler Optimizations

In this paper, we restricted ourselves to finding compiler optimizations that preserve the error resilience of the un-optimized version. However, as we found earlier, compiler optimizations can often enhance the resilience of a program. So we ask what happens if we remove the restriction of simply preserving the error resilience of the program, and attempt to improve the resilience instead through compiler optimizations. To explore this notion, we modified our GA-based algorithm to an Unbounded GA-based algorithm in which the terminating condition is that the average fitness score of the candidates in the population remains constant for numerous generations.

The results are shown in Figure 12. As can be seen, the solutions obtained from the Unbounded GA-based approach generates optimization sequences with resilience either comparable to or only slightly better than the original GA-based approach. Further, the difference in the arithmetic means of the two approaches is only 2%, which is within the error bars of the measurement. This shows that the solutions obtained by our GA-based approach are comparable to the solutions obtained by the unbounded GA-based approach in terms of error resilience. However, the Unbounded GA-based approach took about $2x$ to $20x$ the number of iterations as the GA-based approach. Thus, our original GA-based approach achieves similar results as the Unbounded GA-based approach, but takes much less time.

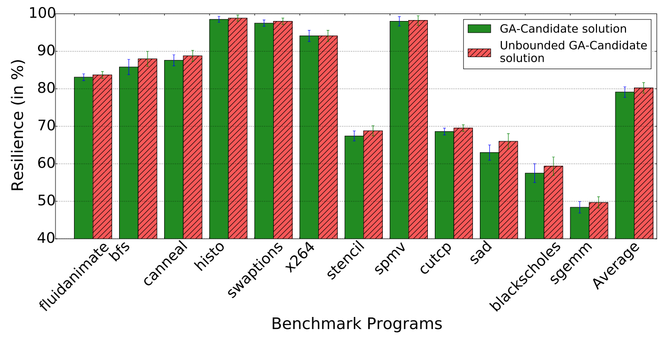

Fig. 12: Resilience of the candidate solutions obtained from GA-based and Unbounded GA-based approaches. Higher values are better.

G. Comparison with Random-Walk

We compare our algorithm with a random walk based approach. In the random walk, we begin with the same individual optimizations that constituted the initial population in our experiments. In every iteration, a random optimization sequence is generated from these individual optimizations and its resilience is evaluated. This process is repeated until a random optimization sequence that does not degrade the program's resilience is obtained. The main difference between the random walk and our approach is that there is no fitness function in the random walk. We observed that the random walk never converged to a solution even after a long time. For example, in the case of the *Blackscholes* program, the random walk did not obtain a solution even after 4 days (230 iterations), whereas our GA-based approach obtained a solution in 4 hours time (24 iterations). Similar results were obtained for other programs.

VIII. Related Work

Demertzi et al. [6] have analyzed the effect of standard optimization levels on the application's vulnerability. There are two differences between their work and ours. First, they do not consider the final outcome of the application due to the error, and whether the error results in an SDC. Second, our technique uses fault injections to evaluate the applications' resilience, while they use ACE analysis [16], which is typically much less accurate than fault injection [31].

Jones et al. [8] have experimentally examined the effect of individual optimization on application vulnerability, and attempt to find a set of optimizations that offer both resilience and performance. Similar to the above study, they also do not consider the final outcome of the application due to the error, and use AVF to measure vulnerability. Further, they do not provide any method to choose the optimal set of optimizations for an application - as we have seen, this set is highly application-dependent.

Sangchoolie et. al. [5] consider the effect of compiler optimization level on the SDC rates of different applications. They find that the optimization levels degrade the resilience of the application (though they argue that this degradation is within acceptable limits). There are two main differences between their work and ours. First, we consider individual optimizations such as *loop invariant code motion*, rather than the standard optimization levels. Secondly, by choosing the individual optimizations, our technique can adapt to the error resilience characteristics of the application.

Thomas et al [7] study the impact of optimizations on error resilience for soft computing applications. They measure the impact of optimization on Egregious Data Corruptions (EDCs), which are significant deviations from fault-free outcomes. There are two main differences between their work and ours. First, EDCs are only a subset of SDCs, and do not apply to general purpose applications where any deviation in the output from the fault-free outcome, no matter how small, is unacceptable. Second, they do not have an automated method to choose the set of optimizations for a given application and platform, relying instead on simple heuristics such as dynamic code size which may not work across different applications.

In very recent work, Cong et al [32] proposed a metric for measuring loop reliability to analyze the impact of loop transformations on software reliability. Similar to Thomas et al., they mainly focus on soft computing applications and EDC outcomes. Further, they restrict their analysis for loop optimizations, while we consider all compiler optimizations.

Rehman et. al. [33] propose novel compiler optimizations that take into account the vulnerability of individual instructions to soft errors. They show that their technique can achieve high reliability for a given performance overhead bound provided by the programmer. Further, Lu et al [21] and Laguna et al. [22] propose a compiler-based technique to identify instructions in the program and selectively protect them. The main difference between these techniques and ours is that they focus on increasing the resilience of programs by transforming the code, thereby incurring a performance overhead. In contrast, our technique focuses on finding compiler optimizations that improve the performance of the code.

Finally, Kanev et. al. [34] have explored the effect of compiler optimizations on voltage noise in microprocessors. While voltage noise can lead to timing violations, we are interested in soft errors that lead to transient faults. Further, they consider the effect of the standard optimization levels only, while we consider the effect of individual optimizations.

IX. Conclusion

In this paper, we reexamined the effect of compiler optimizations on programs' error resilience. Prior work has found that optimizations degrade a programs' resilience; we find that this is not always the case and that some optimizations can improve resilience, but the effects are very application-specific. This demonstrates that resilience and performance optimization need not be odds with each other, and one can improve both by judiciously choosing compiler optimizations.

Based on this insight, we developed a technique that leveraged Genetic Algorithms (GA), a meta-heuristic search approach, to find an optimization sequence that improves the program's performance without degrading its error resilience. We evaluated our technique on twelve benchmark programs, and found that the optimization sequences it generates provide better error resilience than the standard optimization levels (O1, O2 and O3), and often even better than the unoptimized code's resilience, while achieving comparable performance improvement as the standard optimization levels.

For future work, we plan to explore the use of other search heuristics, and to look at other aspects than performance. such as energy consumption.

Acknowledgements

This work was supported in part by grants from the Natural Science and Engineering Research Council of Canada (NSERC), and the Canada Foundation for Innovation (CFI). We thank Behrooz Sangchoolie, Mieszko Lis and the anonymous reviewers of the EDCC 2016 conference for their comments, which have helped us to improve this paper.

References

[1] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *MICRO*, 2005.

[2] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," in *ASPLOS*, 2010.

[3] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Swat: An error resilient system," *SELSE*, 2008.

[4] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[5] B. Sangchoolie, F. Ayatolahi, R. Johansson, and J. Karlsson, "A study of the impact of bit-flip errors on programs compiled with different optimization levels," in *EDCC*, 2014.

[6] M. Demertzi, M. Annavaram, and M. Hall, "Analyzing the effects of compiler optimizations on application reliability," in *IISWC*, 2011.

[7] A. Thomas, J. Clapauch, and K. Pattabiraman, "Effect of compiler optimizations on the error resilience of soft computing applications," in *Workshop on Application and Algorithmic Error Resilience*, 2013.

[8] T. Jonen, M. O'Boyle, and O. Ergin, "Evaluating the effects of compiler optimisations on avf," in *In INTERACT '08*, 2008.

[9] S. R. Ladd, "Analysis of compiler optimizations via an evolutionary algorithm," https://packages.debian.org/sid/devel/acovea.

[10] M. Stephenson, U.-M. O'Reilly, M. C. Martin, and S. Amarasinghe, "Genetic programming applied to compiler heuristic optimization," in *EuroGP*, 2003.

[11] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-compiler software optimization for reducing energy," *ASPLOS*, 2014.

[12] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *LCTES*, 1999.

[13] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT*, 2008.

[15] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *CRHC Technical Report*, 2012.

[16] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *MICRO*, 2003.

[17] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and AI.* 1992.

[18] J. Grefenstette, "Optimization of control parameters for genetic algorithms," *SMC*, Jan 1986.

[19] J. D. Schaffer, R. A. Caruana, L. J. Eshelman, and R. Das, "A study of control parameters affecting online performance of genetic algorithms for function optimization," in *GEM*, 1989.

[20] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *DAC*, 2013.

[21] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "SDCTune: a model for predicting the sdc proneness of an application for configurable protection," in *CASES*, 2014.

[22] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "Ipas: Intelligent protection against silent output corruption in scientific applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, 2016.

[23] K. A. De Jong and W. M. Spears, "An analysis of the interacting roles of population size and crossover in genetic algorithms," in *PPSN*, 1991.

[24] M. Srinivas and L. Patnaik, "Genetic algorithms: a survey," *Computer*, 1994.

[25] J. E. Baker, "Adaptive selection methods for genetic algorithms," in *Proceedings of an International Conference on Genetic Algorithms and their applications*, 1985.

[26] K. Nara, A. Shiose, M. Kitagawa, and T. Ishihara, "Implementation of genetic algorithm for distribution systems loss minimum re-configuration," *IEEE Transactions on Power Systems*, Aug 1992.

[27] B. J. Park, H. R. Choi, and H. S. Kim, "A hybrid genetic algorithm for the job shop scheduling problems," *Comp. & industrial engg.*, 2003.

[28] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *DSN*, 2014.

[29] R. L. Haupt, "Optimum population size and mutation rate for a simple real genetic algorithm that optimizes array factors," in *APSURSI*, 2000.

[30] P. P. Urone, R. Hinrichs, K. Dirks, and M. Sharma, "College physics," 2013.

[31] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ACE analysis reliability estimates using fault-injection," in *ISCA*, 2007.

[32] J. Cong and C. H. Yu, "Impact of loop transformations on software reliability," in *ICCAD*, 2015.

[33] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel, "Reliable software for unreliable hardware: Embedded code generation aiming at reliability," in *CODES+ISSS*, Oct 2011.

[34] S. Kanev, T. M. Jones, G.-Y. Wei, D. Brooks, and V. J. Reddi, "Measuring code optimization impact on voltage noise," in *SELSE*, 2013.