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

by

Nithya Narayanamurthy

B. E., Anna University, 2011

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

October 2015

Abstract

With the projected increase in hardware error rates in the future, software needs to be resilient to hardware faults. An important factor affecting a program's error resilience is the set of optimizations used when compiling it. Compiler optimizations typically optimize for performance or space, and rarely for error resilience. However, prior work has found that applying optimizations injudiciously can lower the program's error resilience as they often eliminate redundancy in the program.

In this work, we propose automated techniques to find the set of compiler optimizations that can boost performance without degrading its overall resilience. Due to the large size of the search space, we use search heuristic algorithms to efficiently explore the space and find an optimal sequence of optimizations for a given program. We find that the resulting optimization sequences have significantly higher error resilience than the standard optimization levels (i.e., O1, O2, O3), while attaining comparable performance improvements with the optimizations levels. We also find that the resulting sequences reduce the overall vulnerability of the applications compared to the standard optimization levels.

Preface

This thesis is based on a work conducted by myself in collaboration with Dr. Karthik Pattabiraman. I was responsible for coming up with the solution and validating it, evaluating the solution and analyzing the results. Karthik was responsible for guiding me on all the core aspects like formalization of the problem, with the solution reasoning, methodology, and analysis and interpretation of results.

Table of Contents

List of Tables

List of Figures

List of Acronyms

LLVM Low Level Virtual Machine

LLFI LLVM based Fault Injector

IR Intermediate Representation

SDC Silent Data Corruption

EDC Egregious Data Corruption

GA Genetic Algorithm

SA Simulated Annealing

CS Candidate Solution

Acknowledgments

I would like to thank my advisor Prof. Karthik Pattabiraman for his support, valuable guidance and encouragement. Karthik has always been motivating me to ponder beyond the norm and guided me in the right direction. He constantly encouraged for my progressive work, at times when I faced difficulties and disappointments. His enthusiasm to share one's thoughts and interrogate on new ideas is something that I find very inspiring.

I would like to thank my lab colleagues with whom I have had many insightful discussions. Their questions have often made me think in diverse directions. Thanks to my colleagues and the professors at the CSRG meetings who helped me expand my knowledge in other research fields.

This would have never been possible without the support from my family and friends. I want to thank my parents, sisters and brother-in-laws for their sincere love and support through this journey. Words cannot do justice to their contribution. Special thanks to my friend Vignesh who has made me strong through this entire journey. I wish to thank all my friends in India who helped and encouraged me to plan my graduate studies. Thanks to all my friends in Canada who made me feel this new country as my home.

Last but not the least, I thank the almighty for providing me with the above!

Dedication

To my parents

Chapter 1

Intoduction

1.1 Motivation

Transient hardware faults (i.e., soft errors) are becoming more frequent as feature sizes shrink and manufacturing variations increase [3]. 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 [10, 19]. This is because the traditional methods of handling hardware faults such as dual modular redundancy and guard banding consume significant amounts of energy, which makes their use challenging for most commodity systems where energy is a first class constraint. Therefore, it becomes critical to design software applications that are resilient to hardware faults.

One of the most important decisions a programmer wishing to build error-resilient applications must make is whether to run compiler optimizations on it. Compiler optimizations, while boosting performance, can often have a deleterious effect on resilience [8, 28, 34] as they remove some of its redundancy. On the other hand, applying optimizations can make programs run faster, thus making them less vulnerable to hardware errors in the first place [1].

In this work, we ask the question: "Do compiler optimizations hurt or improve

[1] We define vulnerability as the probability that an error occurs in the program and leads to a failure, and resilience as the conditional probability that an error leads to a failure given that it occurs in the first place. Thus vulnerability = $(1 - resilience) * executionTime$.

error resilience and vulnerability of programs ?". This question is important for programmers to decide whether to apply optimizations to those programs for which resilience matters. Prior work [8, 28] 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 or vulnerability. While this is useful, the standard optimization levels group together many optimizations, and hence prior work does not disambiguate the effects of individual optimizations on error resilience (and vulnerability). Thomas et al. [34] 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., multi-media applications. To the best of our knowledge, there is no work that has evaluated the effect of *individual* optimizations on the error resilience of general-purpose programs.

1.2 Research Goal and Proposed Solutions

In this work, we first perform an experimental study (Chapter 3) to understand the effect of individual optimizations on program's error resilience. As mentioned, we distinguish between error resilience and vulnerability to separate the effects of compiler optimizations on execution time and code structure. 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 [8, 28], we find that some compiler optimizations can actually improve the error resilience of the program in addition to its performance, thus doubly reducing the program's vulnerability.

Based on this insight, we devise automated techniques to find a sequence of optimizations for a given application that preserves its error resilience. In other words, we attempt to find sequences of individual optimizations for an application that do not degrade the application's error resilience, while improving its performance, thus reducing its overall vulnerability. However, the space of all possible optimizations to consider when optimizing for both resilience and performance is extremely large and brute force search is intractable. Therefore, we leverage meta-heuristic techniques proposed in prior work for performance and memory optimizations [6, 16, 38].

The meta-heuristic search techniques we use in this work are Genetic Algorithms (GA) and Simulated Annealing (SA). Based on our results we suggest GA over Simulated Annealing (SA), as we found that GA was faster and yielded better solutions in our experiments. Applying GAs and SA to the problem of finding resilience-preserving optimizations requires two things. First, we need to develop appropriate operators for the GA and SA to find optimization sequences that satisfy the desired resilience levels. Secondly, we need to come up with a set of parameters for the algorithms to ensure that they converge within a reasonable amount of time. *To the best of our knowledge, we are the first to use a meta-heuristic search algorithms such as GA and SA to find compiler optimization sequences that can improve performance without degrading error resilience.*

1.3 Contributions

We make the following contributions in our work:

- Study the effect of individual optimizations on different programs' error resilience through fault-injection experiments,
- Propose GA-based and SA-based techniques to find a compiler optimization sequence for a given application that does not degrade the error resilience,
- Implement the techniques in a production, open-source compiler, LLVM [17],
- Experimentally tune the parameters of the GA-based and SA-based approaches to achieve fast convergence to solution,
- Evaluate our technique on 12 programs from the PARSEC [2] and Parboil [33] benchmark suites using fault-injection experiments, in terms of its error resilience, performance and vulnerability, and compare it to the standard optimization levels.

The main results of our experimental evaluation are: (1) the resilience of the candidate optimization sequences found by our techniques (GA and SA) is much better than those of the standard optimization levels, and in many cases, even better than that of the unoptimized code, (2) the performance of the optimized code

with our techniques is on par with or only slightly lower than the performance of the code with the standard optimization levels (GA based - better than O1, O2 and sligthly worse than O3 by 0.39%; SA based - 0.61%, 2.27% and 2.72% worse than O1, O2 and O3 respectively), (3) On average, our techniques considerably *lower* the overall vulnerability of the application (GA- 8.12 ($\pm$0.21) and SA- 8.51 ($\pm$0.22) on average), while the standard optimization levels O1 *increase* the overall vulnerability of the application (O1-9.53 ($\pm$0.25) on average) and O2 and O3 reduce it slightly (O2-9.22 ($\pm$0.24) and O3-9.11 ($\pm$0.24)). Thus, for a small performance loss compared to the most aggressive optimization level, our techniques significantly reduces the overall application vulnerability from the unoptimized code.

Chapter 2

Background and Fault Model

In this chapter, we first define error resilience and vulnerability. We then present a brief overview of Genetic Algorithms, Simulated Annealing and then describe our fault model.

2.1 Error Resilience and SDC

A hardware fault can cause a program to fail in one of three ways: it may cause the program to crash, hang, or have an 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 the program to prevent an error that occurs in it 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). This is different from vulnerability, which is the unconditional probability of a fault occurring in the program and leading to an SDC. As mentioned earlier, 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 no-

tion of the Architectural Vulnerability Factor [21], which is defined in terms of the number of bits in a hardware structure that are needed for architecturally correct execution (ACE). We eschew the traditional 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).

In our work, we attempt to choose optimizations that maintain the error resilience of the application compared to the unoptimized version. 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 resilience is maintained the same after the optimization is applied (due to shorter execution time).

2.2 Genetic Algorithm (GA)

A Genetic Algorithm (*GA*) [14] is a search heuristic 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 [11] [29]. 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.

2.3 Simulated Annealing (SA)

Simulated annealing is a meta-heuristic search algorithm that derives its name from the metallurgical process, *"Annealing"* [35]. *Annealing* [9] is a process used to alter the properties of a metal in which the metal is heated to a certain high temperature and then allowed to slowly cool with a specific cooling rate. Similarly, SA explores the search space controlled by variables *T-temperature* and α-*Cooling rate*.

The algorithm starts by selecting a random state as its current state from the given initial set, and the variable temperature T is assigned to a high value. It explores a neighboring state from the current state, and evaluates it by comparing its score with the current state. A neighboring state is evolved by applying some slight modifications to the current state. The score of a state determines how good the state is as a potential solution for a given problem. The probability of accepting a state depends on its score, and the variables T and α. Initially when T is high, the probability of accepting a state with a bad score is also high, thus expanding the scope of the search for finding an optimal solution. T is gradually decreased depending on the cooling rate α as the algorithm progresses, and hence the probability of accepting a weak state also decreases. If a state is accepted, then the algorithm evolves a new stats from the accepted state for the next iteration. This process continues until an optimal solution is obtained.

This algorithm handles the problem of local maxima suffered by Genetic Algorithm as its evolution accepts a good number of bad states (states with a bad score)

and tends to proceed towards global maxima [20].

2.4 Fault Model

Transient hardware 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. Faults occurring in the instructions encoding can be detected by the use of simple codes as the instructions do not have the ability to change over execution while the data can change. 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.

Propagation of an error from the component to the application level due to a particle strike is illustrated in the following example. Let us consider an application *A* that is being executed in the processor exclusively. Consider the snapshot of the processor during a clock cycle n. Instructions i_5, i_4, i_3, i_2, i_1 are currently in the pipeline stages *Fetch, Decode, Execute, Memory, Write back* stages respectively. If the particle strikes a component in the *execute* stage of the pipeline, result of the instruction in that stage - i_3 is compromised and produces erroneous outcome.

The fault model that we consider replicates this physical phenomenon of a particle striking the processor components. Coverage of the chip components by the fault model is given in table 2.1. We simulate a particle strike affecting the covered components in the following way. While the application is in execution, one of its dynamic instruction is picked randomly from an uniform distribution. We perform a bit-flip in any register content of this instruction. This randomness adheres to the uniform probability of a fault impacting any computational component of the processor at a given time. In the given example, the fault occurring at the execute stage of instruction i_3 would result in storing the faulty value in the destination register, and we replicate this by performing a bit flit in the value stored in the

Table 2.1: Coverage of the fault model

Components	**Covered?**
Memory	No
Instruction and Data cache	No
Processor pipeline stages	Yes
ALU	Yes
Data bus	No
Control unit	No
Registers	Partially (Specific target is selected; no uniform distribution)

destination register. We use the single bit-flip model to represent transient faults. Prior work [5] has found that there may be significant differences in the raw rates of faults exposed to the software layer when fault injections are performed in the hardware. However, we are interested in faults that are not masked by the hardware and make their way to the application. Therefore, we inject faults directly at the application level. The assumption in this fault model is that the application of interest is currently being executed in the processor exclusively. If multiple applications are running in the processor and a particle strikes a component where an instruction of one of the applications is running, only that application gets affected. The other applications continue to run unaffected. Hence, this fault model is also independent of the underlying architecture and the number of applications. We consider single bit flips as this is the de-facto fault model for simulating transient faults in the literature. Finally, we assume that at most one fault occurs during an application's execution, as transient faults are relatively rare events. A similar fault model has been used by prior work in this area [11, 20, 22].

2.5 Summary

In this chapter, we defined Silent Data Corruption (SDC) and explained its importance over the other outcomes of transient hardware faults such as crashes and hangs. We then explained our definition of error resilience and vulnerability of

software applications. We also explained the general working of Genetic Algorithms and Simulated Annealing that were adopted in our automated techniques for finding the resilience friendly compiler optimizations. We finally defined the fault model used in our automated techniques to evaluate the error resilience of software applications.

Chapter 3

Study on Compiler Optimizations

In this chapter we perform an initial fault-injection study that analyzes the effect of individual compiler optimizations on error resilience of software applications. The experimental setup and the benchmarks considered here are described later in Chapter 5.

3.1 Compiler Optimizations

There are several optimizations available in standard optimizing compilers. An optimization is a code transformation that seeks to improve some property of the program e.g., its performance, size. Different code versions can be achieved by applying some optimizations more than once and in different sequences. As mentioned earlier, optimizations are often grouped into packages, or levels such as O1, O2, and O3, which are common sequences of optimizations that offer different trade-offs between performance improvement and memory. Programmers can choose to invoke either a predefined optimization level, or individual optimizations (from a set of pre-defined optimizations), when compiling their code. Table 3.1 lists different common kinds of compiler optimizations and their characteristics.

Optimizations can also affect the error resilience of a program [8, 34]. For example, many optimizations attempt to reduce redundant computations in the program in order to improve its performance. Unfortunately, this also has the effect of increasing the proneness of the program to errors that cause SDCs thereby making

Table 3.1: Different types of compiler optimizations and their characteristics

Type	Optimization	Description
Data flow optimizations	Constant propagation (constprop)	Replaces instructions which have only constant operands with a constant value
	Sparse conditional constant propagation (sccp)	Removes certain types of dead code and propagates a constant through the entire program
	Common subexpression elimination(cse)	Eliminates common subexpressions and replaces them with a variable that computes that common subexpression
Loop Optimizations	Loop invariant code motion(licm)	Moves the invariant code within the loop to the loop pre-header if it is not affected by the loop iterations
	Loop Strength Reduction (loop-reduce)	Replaces expensive operations within the loop with simpler operations
	Unroll Loops (loop-unroll)	Eliminates/rewrites an instruction with repeated independent instructions in the loop to increase the program speed
	Unswitch loops (loop-unswitch)	Moves the conditional inside the loop to outside and duplicates the loop in 'if' and'else' clause
Global Optimizations	Inter-procedural Constant Propagation(ipconstprop)	Similar to constant propagation optimization which is applied across procedures
	Inter-procedural Sparse Conditional Constant Propagation(ipsccp)	Inter-procedural variant of sccp (decides on basic blocks, constants and conditional branches)
	Inlining(inline)	Replaces the function call with the body of the function
Others	Global value Numbering (gvn)	Assigns a number value to variables and expressions that are provably equivalent
	Merge Constants (mergeconst)	Merges duplicate global constants to a single shared constant
	Instruction combine (instcombine)	Combines redundant and simple algebraic instructions

them less error resilient. On the other hand, some optimizations can increase the resilience of a program (see below).

3.2 Fault Injection Study

We chose 10 individual optimizations at random from about 50 optimizations available from the LLVM compiler. We performed an initial study to analyze the impact of individual optimizations on the error resilience of two applications from the PARSEC benchmark suite, namely *Blackscholes* and *Swaptions*. We first compiled each program with the chosen ten different optimizations using the LLVM compiler [17].

We performed fault injection experiments on the unoptimized version and the ten different optimized versions of the programs to measure their respective error resilience. Figure 3.1 shows the resilience (in %) of the different optimized versions of the two programs compared to the resilience of the unoptimized version (baseline). As can be seen in the figure, 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 the other. For example, while the *loop-reduce* optimization improves the resilience of *Blackscholes*, it degrades that of *Swaptions*. Therefore, we need an application-specific technique to find optimization sequences that do not degrade error resilience but improve performance for a given application. This is the goal of this thesis.

3.3 Analysis on Individual Optimization

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 below, one optimization that degrades error resilience and another that enhances error 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

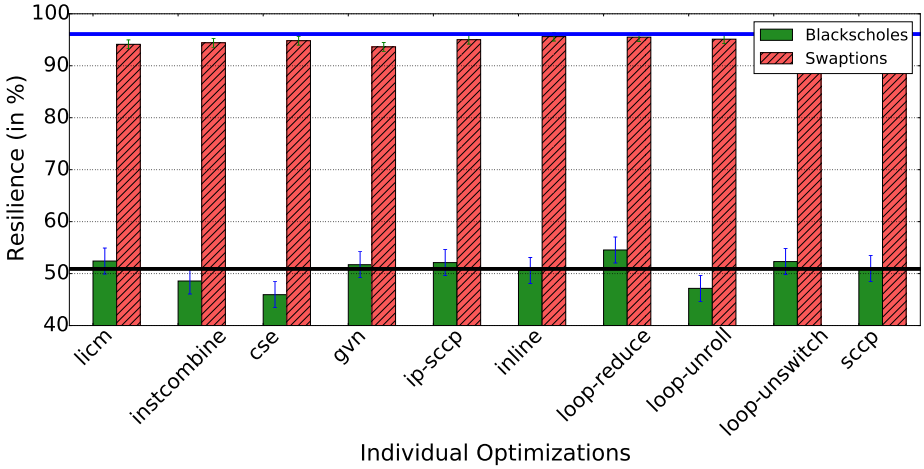

Figure 3.1: Resilience of blackscholes and swaptions optimized with different individual optimizations (Black line represents the resilience of the unoptimized version of blackscholes; Blue line represents the resilience of the unoptimized version of swaptions)

the loop to the pre-header block of the loop without affecting the semantic of the program.

Figure 3.2a shows a code snippet(unoptimized) where the LICM optimization can perform some transformations on the loops and Figure 3.2b 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 3.2a) 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 3.2b.

We performed 3000 fault injections in both the unoptimized and LICM optimized program versions, and observed that the LICM optimization reduces the error resilience of the program.

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 3.2b). 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

```
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 }
```

Figure 3.2: Effect of running the LICM optimization on a code snippet (a) Unoptimized version, (b) Optimized version.

in Figure 3.2a) 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 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 3.3a 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 3.3a) is replaced with temporary variables `temp` and `temp1` for the address and value of array `rs1` as shown in Figure 3.3b (line 6-8). Hence the induction variable here is only used to control the loop entry and exit after the optimization.

As in the previous example, we performed 3000 fault injection experiments in both versions of the programs, and observed that the LOOP-REDUCE optimization enhances the resilience of the program. To understand why the resilience is enhanced in the case of the LOOP-REDUCE optimization, 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

```
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 }
```

Figure 3.3: Effect of running the LOOP-REDUCE optimization on a code snippet (a) Unoptimized version, (b) Optimized version.

resilient that the unoptimized version. This example shows how an optimization can improve the error resilience of an application.

3.4 Summary

In this chapter, we study the behavior of different individual optimizations based on a fault injection study. Our study shows that 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 build an automated method to find optimization sequences that do not lower error resilience of a given application.

Chapter 4

Methodology

In this chapter, we first present the problem statement and discuss its complexity. We then present our GA-based and SA-based approaches for solving the above problem. We finally discuss the implementation details of our approaches.

4.1 Problem Statement and Complexity

We devise automated methods 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. This is why we need an efficient way to search the space of optimizations for resilience, which

is provided by the meta-heuristic search algorithms. While other meta-heuristic search methods are also possible (e.g., Hill Climbing), we use GA and SA as they have been used in prior works on finding compiler optimization sequences for performance and memory.

4.2 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 a chromosome. 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 Chapter 6.

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, 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 devise a fitness function ($\Theta()$) that measures the error resilience of a candi-

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** α_{max}
24: **end procedure**

date optimization sequence. However, as an advanced option any function that measures both performance and resilience/vulnerability can be considered, where the algorithm searches for an optimization sequence that guarantees enormous improvement in performance and resilience. Since we consider only error resilience, the fitness function can be based on a resilience model or on fault injection experiments. We use fault injection for this purpose as we were unsuccessful in finding a fitness function to predict a program's resilience based on its code structure (see appendix). Note however that our method is generic and is not tied to the use of fault injection.

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 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 Chapter 6.

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 [1, 7, 22, 24, 31]. The *MutateRate* was chosen based on our analysis discussed in the Chapter 6, 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 operation. 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 as shown in Figure 4.1. Swap operation is similar to the two-point crossover, where the entities within the two selected index are swapped between the candidates.

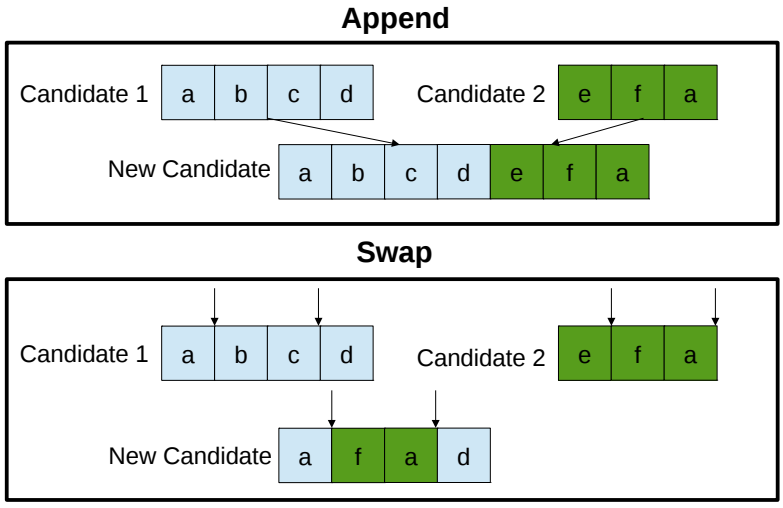

Figure 4.1: Crossover Operations: Entities such as 'a', 'b', 'c' etc are individual compiler optimizations in an optimization sequence.

(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. Figure 4.2 shows adding, deleting or replacing an optimization in the candidate optimization sequence.

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

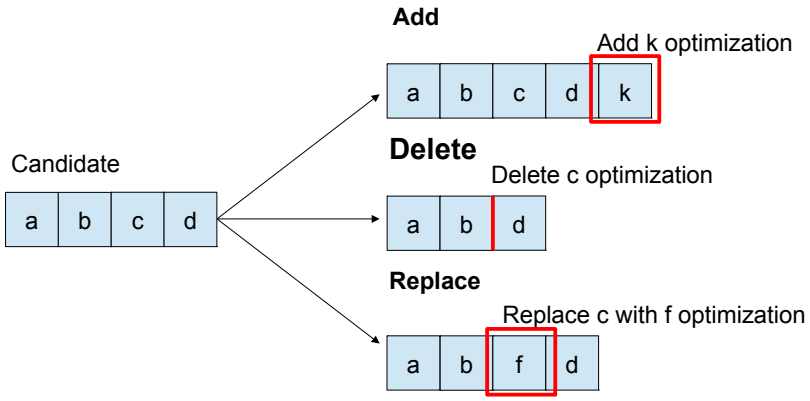

Figure 4.2: Mutation Operations: Entities such as 'a', 'b', 'c' etc are individual compiler optimizations in an optimization sequence.

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.

4.2.1 Representative Example

To understand how a candidate solution (optimization sequence) is obtained by our GA-based algorithm, we illustrate its running with the Blackscholes benchmark program from PARSEC [2]. To simplify the presentation, we present only the steps that were involved in the evolution of the final candidate solution.

1. *Initialization:* We initialized the population with a set of 10 individual optimizations from the LLVM compiler [17]. Let us consider the following as the initial population:{*licm, instcombine, gvn, early-cse, loop-reduce, sccp, inline, ipsccp, loop-unroll, loop-unswitch*} (these optimizations are explained in Table 3.1).

2. *Fitness Function:* The fitness score of every candidate in the population is calculated using a fault injection experiment as explained earlier.

3. *Tournament Selection:* As explained in Section 4.2, the crossover and mutation operations are performed based on the probability values *CrossoverRate* and *MutateRate*. We consider the *Random Selection* strategy for tournament selection.

4. *Unrolling the algorithm:* Assume that two random candidates namely *loop-reduce* and *loop-unroll* are chosen from the population (line 5). A random number is generated and is found to satisfy the *CrossoverRate* probability. Again a random number is generated to perform either a *swap* or *append* operation. In this case, assume that append operation is performed on the two candidates and results in the new candidate *[loop-reduce, loop-unroll]* (lines 6-9). The contents within a square bracket represent a single optimization sequence.

Regardless of whether we perform the crossover operation, a random number is generated to see if it satisfies the *MutateRate* probability (line 11). If crossover was performed, the new candidate undergoes mutation. Otherwise, the first of the two randomly picked candidate undergoes mutation and gives rise to a new candidate. Assume that in this case, the probability does not satisfy *MutateRate*. So *[loop-reduce, loop-unroll]* is the candidate sequence. Fitness score for this candidate is calculated and found to be better than the weakest member, *early-cse*. Hence, the new candidate replaces the weakest candidate in the population. For clarity we have underlined the new candidate added to the population. Now, the population is {*licm, instcombine, gvn, [loop-reduce, loop-unroll], loop-reduce, sccp, inline, ipsccp, loop-unroll, loop-unswitch*} (lines 14-21). This is the end of

the first generation.

In the first generation, the desired target resilience is not yet reached and hence the algorithm proceeds to the second generation. In this generation, *gvn* and *loop-unroll* are chosen as the candidates for selection. In the crossover step, the two candidates give rise to the new candidate, *[gvn, loop-unroll]*. Now assume that the mutation probability happens to be satisfied in the second generation and the replace operation is performed on this candidate to give rise to the new candidate *[gvn, inline]*. Now, the population is {*licm, instcombine, gvn, [loop-reduce, loop-unroll], loop-reduce, [gvn, inline], inline, ipsccp, loop-unroll, loop-unswitch*}.

In the second generation, the target resilience is not yet attained and the algorithm continues. Let us assume the two candidates chosen in the third generation are *[licm, loop-reduce]* and *[gvn, inline]*. Assume that the *CrossoverRate* is satisfied and the two candidates undergo an append operation to yield the new candidate *[loop-reduce, loop-unroll, gvn, inline]*, and that the *MutateRate* is not satisfied in this generation. We find that the resilience of this new candidate satisfies the target resilience and the algorithm terminates. This candidate optimization sequence is presented as the candidate solution that achieves the desired target resilience (line 23). Thus, the algorithm took three generations to obtain the candidate solution in this example. Note that in reality it took many more generations for the program, but we present only three generations to simplify the presentation.

4.3 SA-Based Approach

As in the GA-based approach, we start with an initial set that includes a subset of optimizations available in the compiler. In this approach a *state* is defined as an optimization sequence, while the score of a state is defined as the resilience(%) of the optimized program. Similar to our GA-based approach we consider the resilience of the unoptimized program as the baseline here (i.e., the target resilience).

The SA-based algorithm is shown in Algorithm 2. The step wise execution of the SA-based algorithm is as follows:

1. *Initialization:* We begin by choosing a random individual optimization from the initial set and consider it to be the current state of the algorithm (line 2 of Algorithm 2). The resilience of the program optimized with the optimization(s) in

the current state is measured and it is associated as the score of the current state ($s_{current}$). This score is compared with the target resilience i.e the resilience of the unoptimized program and we terminate the algorithm if the resilience score of the current state is better than or same as that of the target resilience (line 3-4 of Algorithm 2). However, as in the GA-based approach, this trivial condition with an individual optimization was never satisfied in our experiments.

2. *Choosing Neighbor state:* We then generate a neighbor state ($s_{neighbor}$) from the current state (line 5 of algorithm 2) ($ChooseNeighbor()$). The process of generating the neighbor state involves tweaking the current state with specific operations. These operations include add, delete and replace as shown in the Figure 4.3. The add operation involves selecting a random individual optimization from the initial set and appends it to the current state. The replace operation replaces a random optimization in the sequence of the current state with an optimization randomly chosen from the initial set. Finally, the delete operation removes a random optimization from the sequence of the current state.

Current State (Optimization Sequence)	**Example Neigbhor States**
[a b c d e]	Add – [a b c d e x]
	Replace – [a b c y e]
	Delete – [a b d e]

Figure 4.3: Choosing the neighbor states from current state (a, b, c,.. are individual optimizations)

3. *Score Calculation:* Once the neighbor state is generated, the resilience of the program optimized with the optimization sequence in the neighbor state is measured ($s_{neighbor}$) (line 6 of Algorithm 2).In simulated annealing, the score of a state ($ResilienceScore()$) is similar to the fitness score in GA. It determines whether the neighbor state should be carried forward as the current state to the next iteration. Similar to the GA-based approach this score is the error resilience measure of a state. As before we perform fault injections to calculate it.

4. *Accepting neighbor state:* The neighbor state is accepted if its score is better than the score of the current state. However if the neighbor state has a lower

resilience score, the probability of accepting the neighbor state depends on the variables T (temperature) and α (cooling rate) in the algorithm. The probability of acceptance of a weak neighbor state is given by the Equation 4.1 where ΔS is the difference in resilience scores of the current state and neighbor state (line 7-16 of Algorithm 2).

$$P = e^{-\Delta S/T} \tag{4.1}$$

The temperature T is decremented slowly over a period of time (line 18 of Algorithm 2). This cooling down process is determined by the cooling rate α. On every iteration, T is updated based on the Equation 4.2. We later discuss about how the values of T and α are chosen in 6.

$$T = T * \alpha \tag{4.2}$$

As mentioned is Chapter 2 initially when T is high the probability of accepting a weak neighbor state is also high. However as T reduces over iterations, the value of $\Delta S/T$ increases, and hence the probability of accepting a weak neighbor state decreases. Thus initially the probability of accepting a weak state is high and gradually decreases over time in the algorithm. If the neighbor state is accepted, then it will be the current state for the next iteration of the algorithm (line 15-16 of Algorithm 2).

4. *Termination:* The above steps 2 and 4 are repeated until a new neighbor state generated satisfies the terminating condition. As mentioned earlier, the termination condition in our case is the resilience score of the unoptimized program (line 20 of Algorithm 2).

4.4 Measuring Resilience

Both our approaches require us to estimate the error resilience of a candidate optimization sequence in each generation of the algorithm. We attempt to find such a tool based upon program level metrics. If strong correlations exist between a metric 'X' and error resilience, we can then estimate the resilience of a candidate based on measuring the metric 'X', rather than doing time consuming fault-injection ex-

Algorithm 2: SA-based approach to find an optimization sequence that does not degrade error resilience

$\delta \leftarrow$ [Individual optimizations]
$T \leftarrow$ Initial Temperature
$\alpha \leftarrow$ Rate of cooling
$P \leftarrow$ Acceptance Probability
$s_{target} \leftarrow$ Resilience of unoptimized
Input : Source code, Set of individual optimizations
Output: Optimization sequence that does not degrade resilience

```
1:  procedure OPTIMIZATION SEQUENCE FOR RESILIENCE
2:      μ_current = Random(δ)
3:      s_current = ResilienceScore(μ_current)
4:      while s_current <= s_target do
5:          μ_neighbor = ChooseNeighbor(μ_current)
6:          s_neighbor = ResilienceScore(μ_neighbor)
7:          if s_neighbor < s_current then
8:              ΔS = s_current − s_neighbor
9:              P = e^(−ΔS/T)
10:             if Random(0,1) < P then
11:                 μ_current = μ_neighbor
12:                 s_current = s_neighbor
13:             end if
14:         else
15:             μ_current = μ_neighbor
16:             s_current = s_neighbor
17:         end if
18:         T = T * α
19:     end while
20: return μ_current
21: end procedure
```

periments. However, as we will see, determining such factors is very difficult, and we could not find evidence of such correlations for SDCs based on the factors identified in prior work [25, 34] for other kinds of failures. This is why we decided to use fault injections.

We consider two kinds of program metrics to measure the correlation with SDC rates of programs. The first corresponds to instruction counts of the program

that were proposed by Thomas et al. [34]. The second corresponds to data-flow metrics that were suggested by Pattabiraman et al. [25]. Although neither paper advocates the use of these metrics for SDCs, we wanted to measure how good these metrics were for predicting SDCs in an application. We performed the fault injection experiments using the experimental setup and benchmarks described in Chapter 5.

Thomas et al [34] have found a correlation between the dynamic instruction count of a program and its resilience for soft-computing applications, i.e., those applications that have relaxed correctness requirements. We wanted to see if this applies to general-purpose applications too.

Figure 4.4A plots total dynamic instruction count versus the SDC rate for Blackscholes, one of the programs in the PARSEC benchmark suite [2] for 10 different optimizations. It can be seen from the figure that total dynamic instruction count has poor correlation with SDC rate. Hence, dynamic instruction count is not a reliable indicator of the SDC rates of an application.

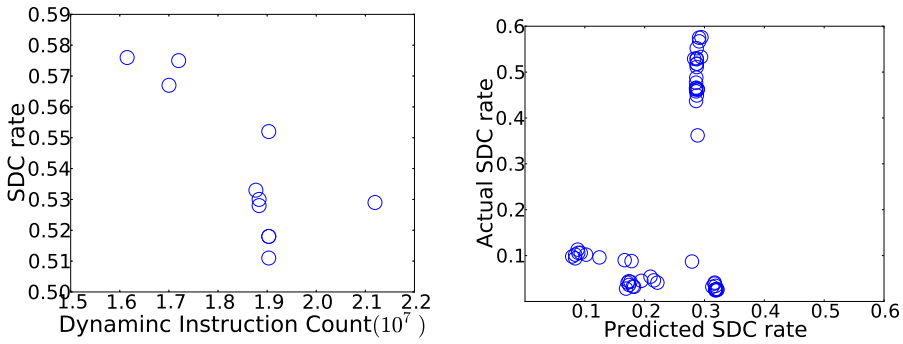

Figure 4.4: (a) Correlation between Total Dynamic instruction count vs SDC rate (for Blackscholes), and (b)Data-flow Regression Model's Estimation vs Actual SDC rate

Pattabiraman et al. [25] have found that data-flow metrics such as fan-outs can be used to predict applications' vulnerability to failures. Based on this intuition, we investigated the following data flow analysis metrics (i) static backward slicing (ii) fan-in and fan-out of variables (iii) loop dependency. Using these factors, we tried to fit a linear regression model that estimates the SDC rate. Figure 4.4B plots

the estimated SDC rate by the regression model against the actual SDC rate for 5 benchmark programs with 10 different compiler optimizations. As can be seen from the figure, the estimated SDC rate by the model has very poor correlation with the actual SDC rate (a similar result was obtained by Hari et al [12]). Therefore, it is nontrivial to determine the error resilience of the program using these factors, and hence we use a fault injection experiment for fitness function and score calculation in our approaches.

4.5 Summary

In this chapter, we described our problem statement and its complexity, motivating the need for automated techniques to solve this problem. We then described our GA-based and SA-based approaches, explaining the step by step process involved in both algorithms. We followed up explaining our GA-based approach with a representative example to have a better understanding on its evolution, obtaining an optimization sequence that does not degrades resilience. Since both our approaches requires us to measure the error resilience of an optimized program, we attempted to find a model/tool measuring the resilience of a program based on its program level metrics. Since we were unsuccessful in finding such a model/tool, we used a fault injection tool for this purpose. In the following chapter, we present our experimental setup for evaluating our approaches.

Chapter 5

Experimental Setup

In this chapter, we first present the implementation details of our techniques and then describe the benchmarks used for our evaluation. We later explain how we tune the GA and SA parameters, and evaluate the resilience, vulnerability and performance overhead of the program compiled with our candidate solutions.

5.1 Implementation

We implemented our approaches using the LLVM compiler [17] , the LLFI fault injection tool [37], and the Java language. LLVM is a popular optimizing compiler that includes a host of standard optimizations, and is used in a wide variety of real-world platforms (including the Mac platform). LLVM allows us to specify individual optimizations to be run on the application, as well as the standard optimization levels. For seeding our techniques, we pick a subset of optimizations consisting of data-flow, loop, global and a few other optimizations available in the LLVM compiler [17]. This subset comprises around 10 different optimizations (we explain why we choose 10 in Chapter 6).

For the fitness function and score evaluation in our approaches, we use LLFI, a fault injection tool that operates at the LLVM Intermediate Representation (IR) code level [17] , 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 [37]. 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 injects specific faults (in our case single bit flips) to the specific instruction operand. At each run the compiled program is executed, and LLFI randomly chooses a single dynamic instance of the instrumented instructions to call its fault injection function which executes fault injected instruction. We then compare the outputs of the fault injection experiments with the fault free outcome to measure the error resilience of the program. Since hardware faults are random occurrences, LLFI randomly selects a dynamic instruction at runtime for fault injection.

5.2 Benchmarks

We evaluate our techniques on twelve programs, five from the PARSEC [2] and seven from the Parboil [33] benchmark suites. The benchmarks represent a wide variety of tasks ranging from video processing to high-performance computing, and are all written in C/C++. They range in size from a few hundred to a few thousand lines of code. The benchmarks chosen and their characteristics are shown in Table 5.1.

5.3 Tuning of the GA parameters

We first evaluate the performance of the GA approach in order to tune its parameters to obtain faster convergence. One way to measure performance of 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. 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 in order to tune the GA. These parameters are explained in Section 4.2.

(1) *MutateRate*: We vary this value based on what the literature on GA recom-

Table 5.1: 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

mends [13], 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) considered 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.

5.4 Tuning of the SA parameters

Similar to the GA-based approach we also tune the SA parameters to evaluate the performance of the SA-based approach for faster convergence. We know that initially the value of T should be high, so that the probability of accepting weak solutions is high. Choosing the initial value of T depends on the possible values of P (probability of acceptance). We observed that the least possible value of ΔS is 0.1. Hence the possible maximum value of T for which P starts from 0.99 is *T=10*.

Similar to tuning GA parameters, we tune the SA parameter *cooling rate* by varying its values with 0.5, 0.7, 0.9 and 0.99. We choose *cooling rate* values from 0.5 as T would decrease rapidly when α is low($T = T * \alpha$).

5.5 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 Chapter 5.1). 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), in order to get tight confidence bounds on the SDC rate. The error bars range from 0.85% to 2.501% depending on the benchmark for the 95% confidence interval. In each run, our fault injector injects a single bit flip into the result of a single dynamic instruction chosen at random from the set of all instructions executed by the program. This is in line with our fault model in Section 2.4.

We compare the results of our techniques with standard optimization levels O1, O2 and O3 as no other prior work has proposed an algorithm for selecting optimizations for resilience. We repeat the above process for each of the optimization levels O1, O2, and O3, for each benchmark program, and obtain their error resilience. We then run our GA-based and SA-based approaches to identify a candidate solutions (i.e., optimization sequence) for each benchmark program, and

then repeat the same experiment for these candidate solutions. We compare each of the resilience values to the baseline resilience of the unoptimized version, and measure the increase or decrease in error resilience with respect to the baseline. Also in order to be accurate, we consider the error bars into account on every iteration of the algorithm while comparing the resilience of the optimization sequences with the target resilience.

There are a total of 72000 injections performed in our experiments (12 benchmarks, 1000 injections, 6 executables namely, O1, O2, O3, unoptimized and GA and SA candidate solutions). Note that these fault injection experiments do not include the overall fault injections performed by the GA-based and SA-based algorithms to evaluate a candidate solution, which are many more in number.

5.6 Performance Evaluation

We then evaluate the performance improvement obtained by each of the optimization levels and the candidate solutions obtained by our approaches. We measure the execution time of the executable compiled with the appropriate set of optimizations i.e., O1, O2, O3 and the GA and SA candidate solutions 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 for each configuration to obtain tight error bars ranging from 0.28% to 3.38% for the 95% confidence interval.

5.7 Summary

In this chapter, we discuss the implementation details of our techniques and also explain the working of LLFI (fault injection tool), that was used in our techniques for measuring the error resilience of a given program. We present the twelve benchmarks used in our evaluation and also discuss about tuning the various algorithm parameters to improve the performance of both approaches. We then discuss our evaluation of the candidate solutions obtained from our approaches with respect to performance and resilience, compared with the standard optimization levels (O1, O2 and O3).

Chapter 6

Results

We first present the results of how we tune the parameters of our GA-based and SA-based algorithms for faster convergence. We then present the results for evaluating the error resilience of the candidate solutions (i.e., optimization sequences) found by our approaches, and that of the standard optimization levels. Finally, we present the results of the performance improvement and vulnerability reduction of each of these optimization sequences over the unoptimized version.

6.1 Effect of GA Parameters

We consider the effect of four parameters on the GA-based approach's convergence rate. In these experiments, we only present the results for two benchmarks, *bfs* and *blackscholes* due to space constraints, but the results we obtained were similar for the other programs.

6.1.1 Mutation Rate

We first considered the effect of varying the mutation rate of the GA-based algorithm, keeping the other parameters fixed. As explained in Chapter 2, 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 6.1 shows the effect of varying the mutation rate on the convergence

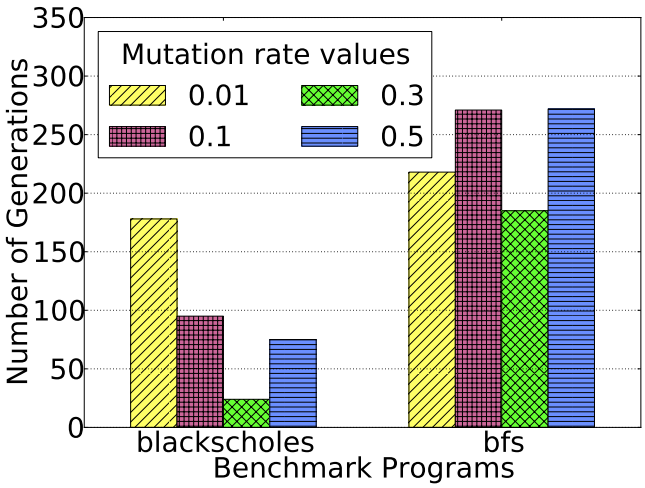

Figure 6.1: Number of generations taken to generate the candidate solution with different mutation rate values.

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.*

6.1.2 Selection Strategy

As mentioned in Section 4.2, there are two possible selection strategies in each iteration of our GA-based 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 is shown in Figure 6.2. It can be observed that for the *blackscholes* benchmark, the score based selection method takes many more generations than the random selection to obtain the candidate solution. The difference is much lesser for the *bfs benchmark*. 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.*

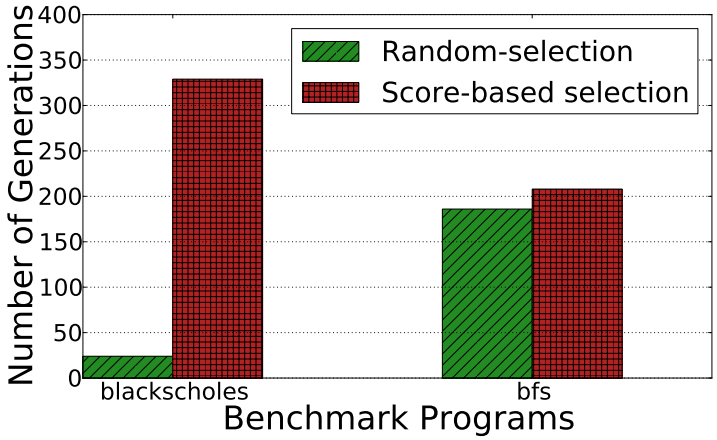

Figure 6.2: Number of generations taken to generate the candidate solution by the random selection and score based selection strategies.

6.1.3 Population Size

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.

Figure 6.3 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 an optimal solution faster. On the other hand, increasing the population size may 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 6.4). *Based on our results, we choose a population size of 10 for our GA-based approach.*

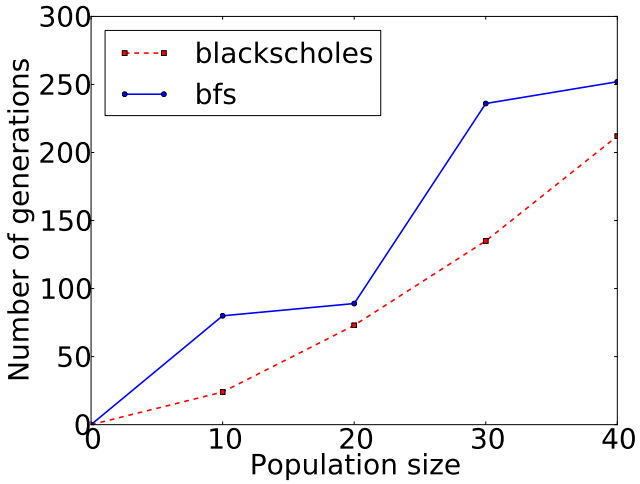

Figure 6.3: Number of generations taken to generate the candidate solution with different population sizes.

6.1.4 Optimization Types

Compiler optimizations are classified into different types based on the transformation they perform on the program. As described in Chapter 3, 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 6.4. 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.

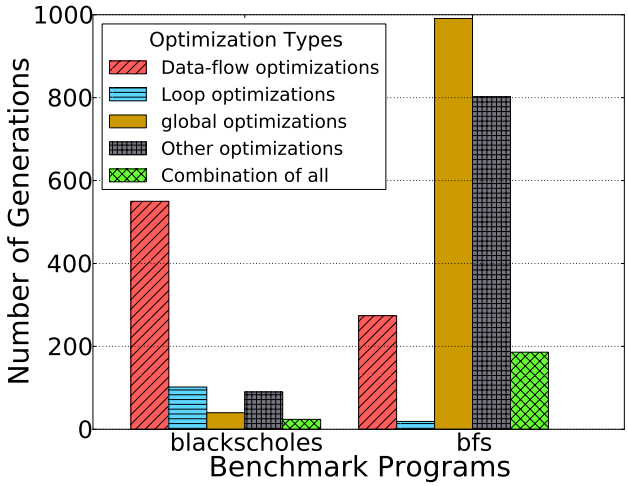

Figure 6.4: Number of generations taken to generate the candidate solution with different optimization types.

6.2 Effect of SA parameters

6.2.1 Rate of Cooling

Rate of cooling is the rate at which the temperature is reduced over iterations in SA. The goal of the algorithm is to obtain an optimal solution by accepting more number of bad states initially, and this can be achieved by reducing the temperature slowly over iterations. Since $T = T * \alpha$ and α lies between 0-1, α should be high for reducing T slowly. However to choose a suitable value of α we evaluate the time of convergence of the algorithm for different α values.

Figure 6.5 shows the effect of varying *alpha* on the convergence rate. We experimented with four different values: 0.5, 0.7, 0.9 and 0.99. We choose values from 0.5 because any value below 0.5 would decrease the temperature rapidly. We observe that our SA-based approach obtains the candidate solution in the least number of iterations is for *alpha* = 0.99. *Therefore, we choose alpha as 0.99 for our SA-based approach.*

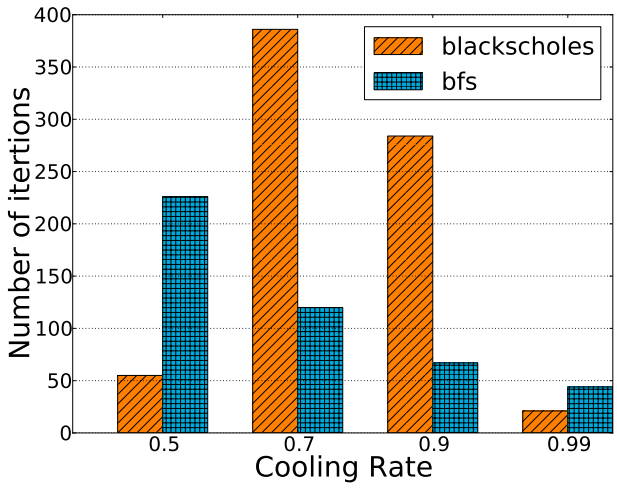

Figure 6.5: Number of iterations taken to generate the candidate solution by the SA-based approach with different cooling rate.

6.3 Resilience Evaluation

Figure 6.6 shows the aggregate fault injection results across benchmarks for the unoptimized, original versions. The percentages of SDCs across all benchmarks is 19.0%. Crashes constitute 36.8% and benign injections constitute 44.2%. 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, as this is line with the definition of resilience (see Section 2.1).

We compare the resilience of the program compiled with the optimization sequence (i.e., candidate solution), obtained from our GA-based and SA-based approaches, with the resilience of the unoptimized program, and that of the compiler optimization levels O1, O2 and O3. Figure 6.7 shows the resilience (in %) of the unoptimized, candidate solutions (from GA and SA) and the different optimization levels.

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 solutions (i.e., optimization sequence) generated by our techniques (GA and SA approach) provides a re-*

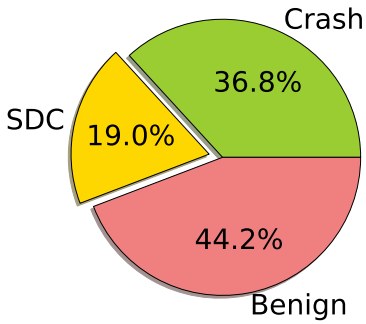

Figure 6.6: Aggregate percentage of SDC, crash and benign results across benchmarks for the unoptimized version.

silience better than or on par with the unoptimized version of the program. It can be observed that for most benchmarks (except *x264*, *cutcp* and *blackscholes*) the candidate solution obtained from the GA-based approach provides better resilience than the SA-based approach. The reason is as follows: an optimization sequence evolved in an iteration in the GA-based approach cumulatively depends on the current population that contains a set of best candidates (optimization sequences with better resilience) evolved until the previous iteration. Whereas in the SA-based approach, an optimization sequence evolved in an iteration depends only on one optimization sequence that was evolved in the previous iteration. That said, the resilience achieved by the candidate solutions obtained from the two techniques do not differ significantly.

The arithmetic mean of the error resilience across benchmarks, of the unoptimized version and the GA-candidate solution, SA-candidate solution, O1, O2 and O3 levels are respectively 76.14 ($\pm$1.54), 79.125 ($\pm$1.57), 78.36 ($\pm$1.55), 72.77 ($\pm$1.6), 73.15 ($\pm$1.6) and 73.38 ($\pm$1.54).[1] Further we show the SDC rates of the unoptimized, candidate solutions (GA and SA) and the optimization levels in Figure 6.8. The arithmetic mean of the SDC rate across benchmarks show that our candidate solutions lower the SDC rate significantly compared to the unoptimized version, O1, O2 and O3.

[1] We use the arithmetic means (AMs) for computing the averages as we are comparing the absolute values.

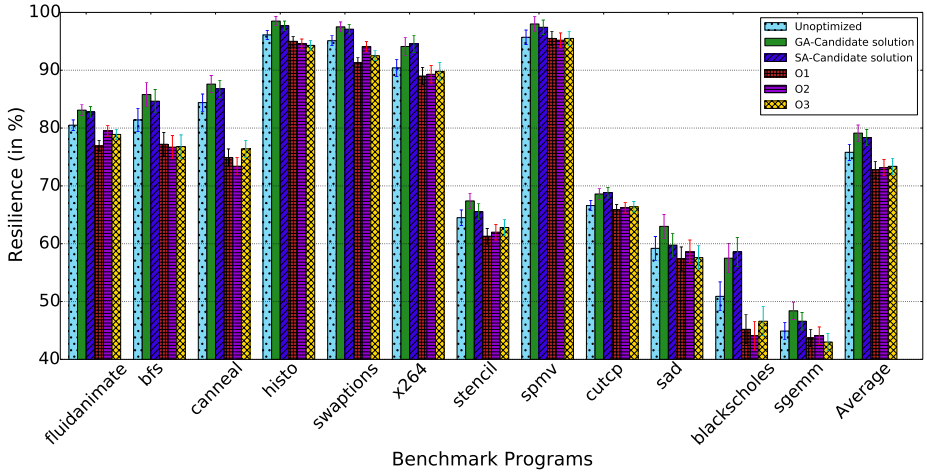

Figure 6.7: Resilience of the Unoptimized, candidate solutions, O1, O2 and O3 levels. (Higher values are better).

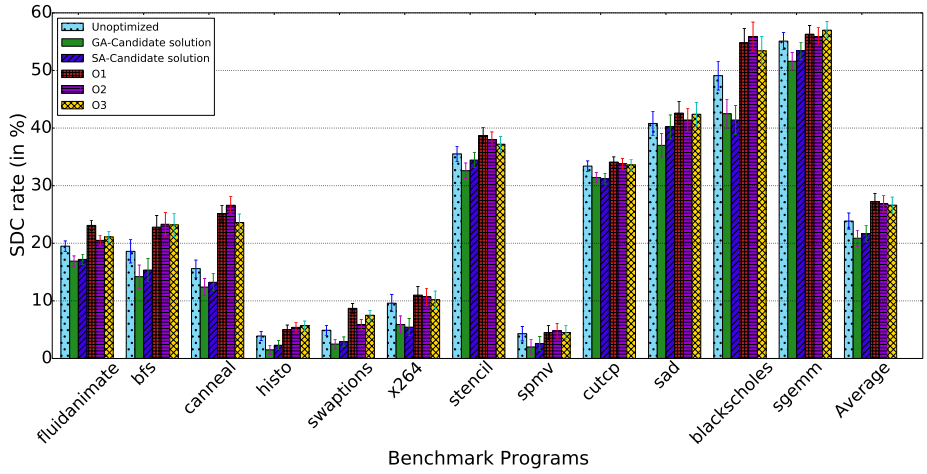

Figure 6.8: SDC rate of the unoptimized code, candidate solutions, O1, O2 and O3 levels. (Lower values are better)

6.4 Performance Evaluation

As in the resilience experiment, we compare the performance of the unoptimized version of the program with the candidate solutions, O1, O2 and O3. Figure 6.9 shows the execution time (in sec) of the unoptimized code, candidate solutions

found by our techniques (GA and SA) and the optimization levels O1, O2 and O3. The figure shows that the candidate solution from our techniques provides better performance than the unoptimized version (as expected). Also we observed, our GA-candidate solutions' average of execution time across benchmarks is better than those of the optimization levels O1 and O2, and slightly worse than O3 (by 0.39%). Similarly, the SA-candidate solutions' average of execution time is 0.61%, 2.27% and 2.72% worse than O1, O2 and O3 respectively. *This shows that resilience friendly optimizations obtain performance improvements that are comparable to the standard optimization levels.*

In fact, in some cases, the performance of the candidate solutions found by our approaches is better than the performance provided by the optimization levels. For example, in the case of the *blackscholes* program, our optimization sequences (both GA and SA) provides a 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 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. Similar behavior was observed in *canneal*. It can also be observed that in some programs such as *swaptions, cutcp, sad* and *sgemm* the candidate solution obtained from SA-based approach do not provide significant performance improvement as compared to that provided by the candidate solution from GA-based approach (error bars do not overlap). In the remaining programs considered the candidate solutions obtained from SA-based approach and GA-based approach provide performance improvement with overlapping error bars. Hence it is difficult to decide on one approach to be better than the other.

Note that performance improvement is not an explicit goal of our techniques, though it is implicit in the fact that we choose to use standard compiler optimizations that are predominantly focused on performance improvement.

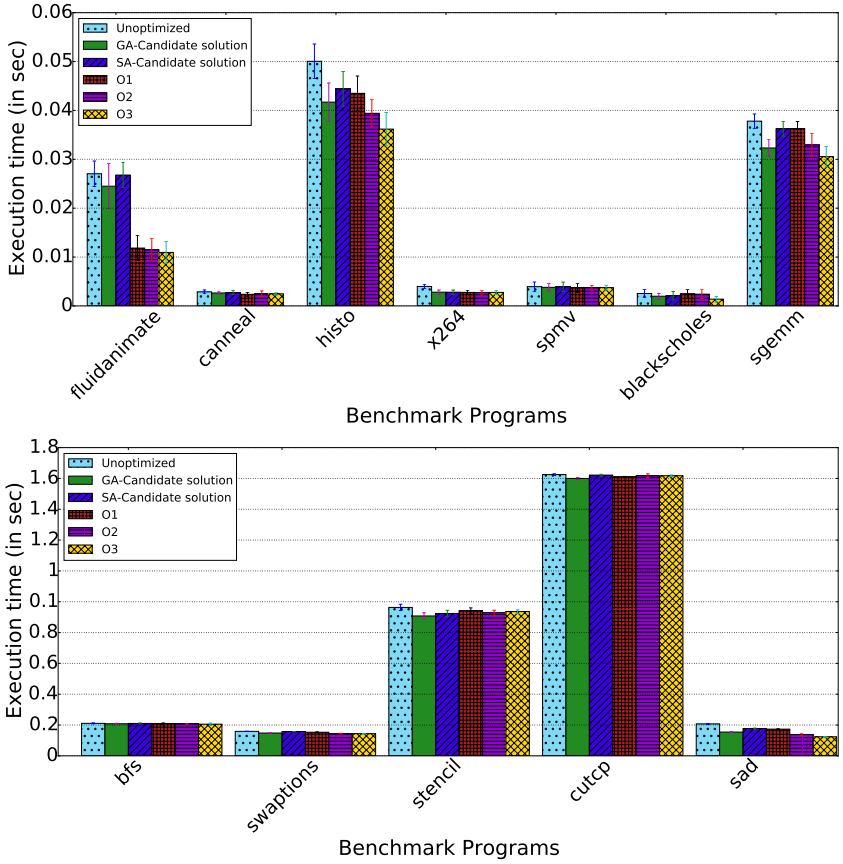

Figure 6.9: Runtime of the unoptimized code, candidate solutions, O1, O2 and O3 levels. (Lower values are better).

6.5 Vulnerability Evaluation

Compiler optimizations often reduce run time of the program, thereby reducing its time of exposure to transient faults. Hence it is important to evaluate the vulnerability of the program in addition to its resilience (Chapter 2). Since vulnerability is calculated as the product of execution time and SDC rate, it is dependent on both these factors.

Similar to the above evaluations, we compare the vulnerability of different versions of the program. Figure 6.10 shows the comparison of vulnerability of the candidate solutions (GA and SA) with optimization levels O1, O2 and O3. Er-

ror bars for vulnerability (product of SDC rate and execution time) is calculated by adding the relative errors which is the standard way for uncertainty calculation for a product [27] The figure shows that in most of the benchmark programs (except *fluidanimate*, and the O3 level in both *sad* and *blackscholes*), the candidate solutions obtained from our techniques reduces the overall vulnerability of the program. On the other hand, the optimization levels O1 and O2 increase the overall vulnerability of the program, while O3 reduces it but not to the level of our candidate solutions. *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). In comparison, the candidate solutions found by our GA-based and SA-based approaches lower the vulnerability to 8.12 (±0.21) and 8.51 (±0.22) than the unoptimized version (9.25 (±0.25).* Since the candidate solutions obtained from our GA-based approach provides resilience and performance improvement better than the SA-based approach, the GA-based approach provides better vulnerability reduction as well. However, in the case of *x264* since SA-based approach provides better resilience and performance improvement, it considerably lowers the vulnerability.

In the case of *fluidanimate*, the standard optimizations reduce vulnerability much more than our candidate solutions. This is because the 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 massive reduction in execution time in Section 6.4.

In the case of *blackscholes* and *sad*, both 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 outweighs the increase in the SDC rate, resulting in lower vulnerability.

6.6 Summary

In this chapter, we first evaluated the performance of our techniques by varying the values of the mentioned GA and SA parameters. Based on the results, we chose the suitable values for the rest of our experiments. We then evaluated the performance, error resilience and vulnerability of the candidate solutions obtained from our techniques and that of the standard optimization levels (O1, O2 and O3). We observed

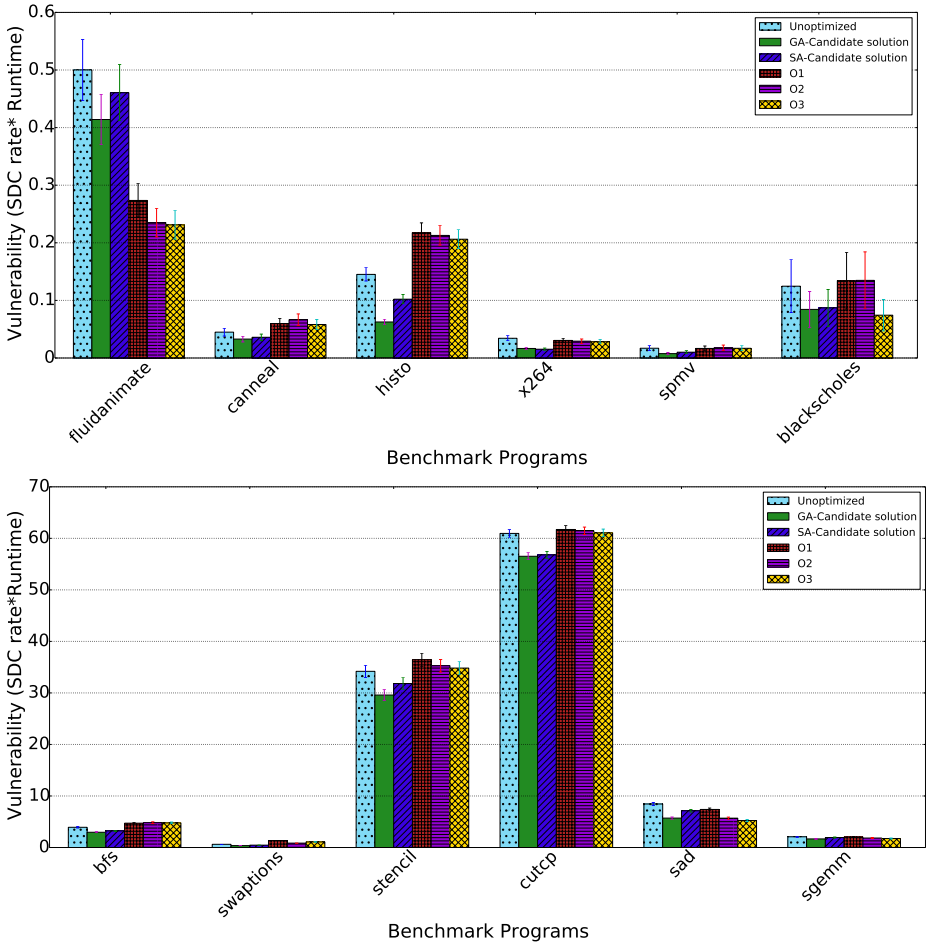

Figure 6.10: Vulnerability of the unoptimized code, candidate solutions, O1, O2 and O3 levels. (Lower values are better).

that the resilience of the candidate optimization sequences found by our techniques (GA and SA) is much better than those of the standard optimization levels. Also the performance of the optimized code with our techniques is on par with or only slightly lower than the performance of the code with the standard optimization levels (GA based - better than O1, O2 and slightly worse than O3 (by 0.39%); SA based - 0.6%, 2.27% and 2.72% worse than O1, O2 and O3 respectively). Finally on an average, our techniques considerably *lower* the overall vulnerability of the application (GA- 8.12 ($\pm$0.21) and SA- 8.51 ($\pm$0.22) on average), while the stan-

dard optimization levels O1 *increase* the overall vulnerability of the application (O1- 9.53 ($\pm$0.25) on average) and O2 and O3 reduce it slightly (O2-9.22 ($\pm$0.24) and O3-9.11 ($\pm$0.24)). In the following chapter, we perform further analysis based on our results.

Chapter 7

Discussion

In this chapter, we examine the implications of our results. We first compare and analyze the candidate solutions obtained from our GA-based and SA-based approaches. Based on our analysis, we recommend that GA-based approach is more suitable for our problem statement, and hence we perform further analysis on the results obtained from the GA-based approach to understand its sensitiveness. We finally reflect on the limitations of our approaches.

7.1 GA-based Vs SA-based

We compare the candidate solutions obtained from our two approaches and analyze whether the resilience improvement is caused by the common individual optimizations they share. We do this analysis to group those individual optimizations as the resilience-friendly optimizations and these optimizations can be directly considered for finding the optimization sequence.

Given that we have two candidate solutions obtained from different approaches for a program, we compare them to examine whether there exists a fixed set of individual optimizations that provides better performance and resilience. For this purpose, we analyzed the common individual optimizations in the candidate solutions obtained from the GA-based and SA-based approaches for each program. We evaluate the impact of these individual optimizations on the resilience of the program, and observed that the resilience improvement of the candidate solutions is

not because of the individual optimizations, but is the effect of the optimization sequence as whole. For example, we extracted the common individual optimizations *loop-reduce* and *gvn* from the candidate solutions obtained by the GA-based and SA-based approaches for the *blackscholes* program. Figure 7.1, shows that *loop-reduce* and *gvn* do not provide such high resilience improvement as rendered by the candidate solutions. This shows that the resilience improvement provided by the candidate solutions is caused by the entire optimization sequence. The transformation performed by *loop-reduce* and the explanation for how it improves the error resilience has already been explained in Chapter 3. In the case of *gvn*, the resilience improvement lies within the error bar, and hence we cannot conclude anything about its resilience improvement.

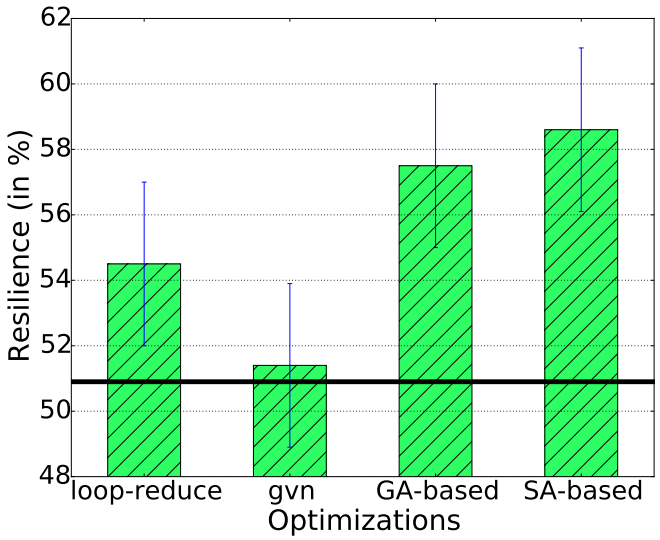

Figure 7.1: Resilience of blackscholes with loop-reduce, gvn and candidate solutions obtained from GA-based and SA-based approaches with the unoptimized code's resilience as baseline. Black line represents the resilience of the unoptimized code.

7.2 Sensitivity Analysis

We compare the resilience improvement rendered by the candidate solutions obtained from our GA-based and SA-based approaches. Based on our results we

observed that it is difficult to decide one among them as the better approach for finding resilience friendly compiler optimizations. This is because we notice that for some benchmarks GA-based approach performs better, while for others SA-based approach performs better. However the candidate solutions obtained from our GA-based approach provide performance improvement that is 5.5% better than SA-based. Also, it depends on the optimization sequences obtained from the two approaches. For example the candidate solution for blackscholes from GA-based approach included *inline*, while that from SA-based approach did not. Analyzing the optimized code we observed that the *inline* optimization has a major contribution to the performance improvement in the candidate solution obtained from the GA-based approach. Since the candidate solution from SA-based approach did not contain *inline* or any other similar optimization that improves performance comparably, it does not provide as good performance improvement as GA-based, O1, O2 or O3 optimization levels. Hence our further analysis in this chapter is based on the results of the GA-based approach.

7.2.1 Order of the optimizations

We evaluate the sensitivity of error resilience with the order of individual optimizations in the optimization sequences (candidate solutions) obtained from our techniques. We measure the resilience of the program optimized with all possible permutations of optimizations in the sequence. If there are 'n' individual optimizations in the sequence, then we have n! sequences to be validated. For example, the candidate solution obtained from GA-based approach for the *blackscholes* program includes 4 individual optimizations *[loop-reduce, loop-unroll, gvn, inline]*, and hence we have 24 sequences. Figure 7.2 shows the resilience of the candidate solution (CS) obtained from the GA-based approach and the different sequences (we denote different sequences as C1,C2,...C23). In some cases like CS and C4, the order of the optimizations in a sequence is sensitive to resilience (beyond error bars). However, in others since the error bars of the resilience measurements overlap, we cannot draw such a definitive conclusion that the resilience of the program is sensitive to every change in the order of the optimizations.

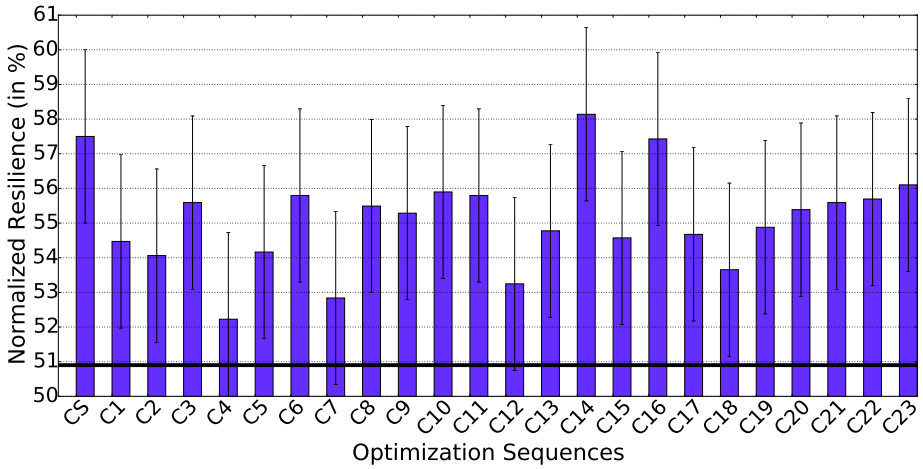

Figure 7.2: Resilience of blackscholes with the candidate solution-GA based (CS) and the different combinations of the sequence (C1,C2,....C23) with the unoptimized code's resilience as baseline.

7.2.2 Resilience vs Vulnerability measuring fitness function

We evaluate the candidate solutions obtained by the GA-based approach, varying its fitness function for resilience and vulnerability measurement. The goal of the algorithm with the vulnerability measuring fitness function is to find an optimization sequence that does not increase the vulnerability of the program. Figure 7.3 compares the vulnerability of the candidate solutions (GA-based) with resilience and vulnerability measuring fitness functions, O1, O2 and O3. It shows that the vulnerability improvement rendered by the candidate solutions obtained with vulnerability measuring fitness function is comparable to those of candidate solutions obtained with resilience measuring fitness function. However, in most cases the vulnerability reduction of the candidate solutions obtained with the vulnerability measuring fitness function was due to its performance improvement and not due to resilience improvement. On the other hand, our GA-based algorithm with resilience measuring fitness function improves resilience and performance reasonably,thus improving vulnerability. This shows that the algorithm with vulnerability measuring fitness function mainly optimizes to generate optimization sequence that improve performance which is another way of improving vulnerability. Also, if the

algorithm with resilience measuring fitness function took x iterations to obtain the candidate solution the vulnerability measuring fitness function took about $1.2x$ to $10x$ iterations depending on the application. This is because the algorithm with vulnerability measuring fitness function tries to optimize for both, time and resilience. *Hence the resilience measuring fitness function is more suitable than vulnerability measuring fitness function.*

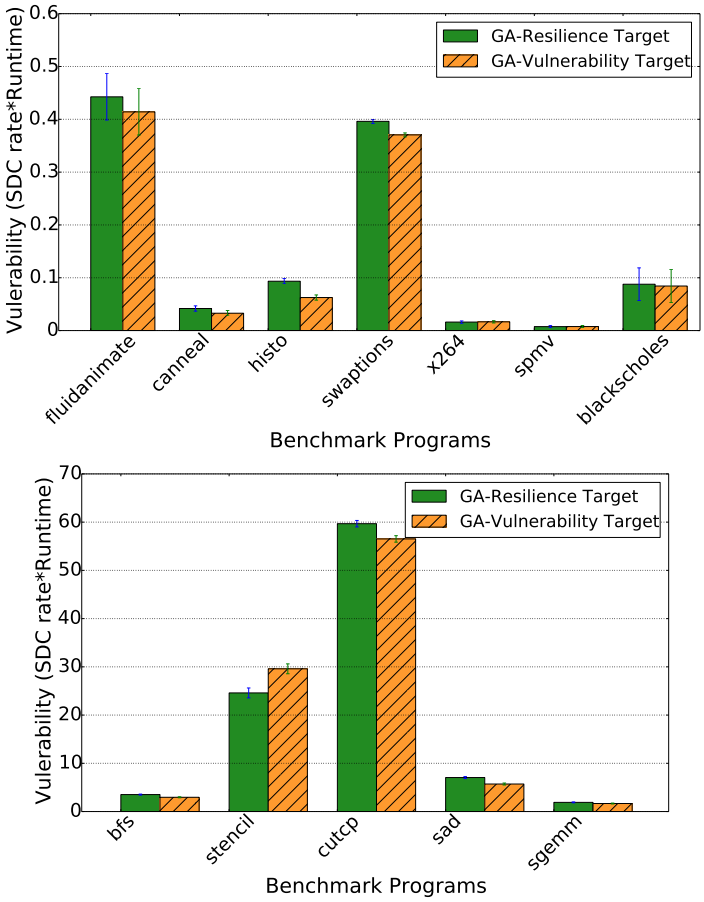

Figure 7.3: Vulnerability of the candidate solutions with resilience and vulnerability target. Lower values are better.

7.2.3 Evolution of the GA-based approach

We now examine how our GA-based approach achieves its objectives. We first compare our algorithm with a random walk search heuristic. 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 an 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.

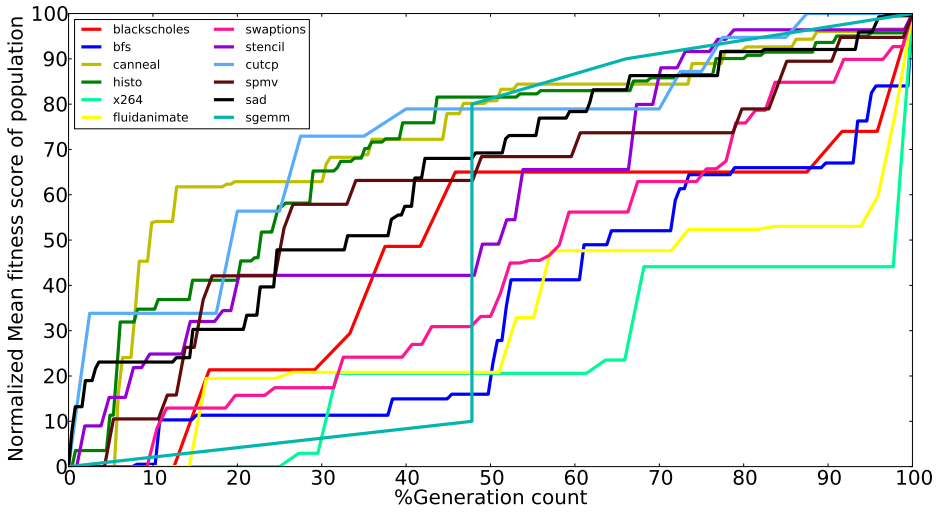

Figure 7.4: Mean population fitness scores during the GA evolution process for each program.

Thus, the main feature of the GA-based approach that allows it to find the target optimization sequence is its evolution, i.e., with every generation, new candidates with better fitness score are evolved. Figure 7.4 shows the mean fitness score of

the population during the lifetime of GA as it progresses from one generation to another for each of the benchmarks. To ensure uniformity across benchmarks, we normalize the number of generations to the total number of generations taken by the GA to evolve the candidate solution for the benchmark. We do the same for the error resilience by normalizing it to the final resilience of the candidate solution. As can be seen from the figure, the fitness score of the population (i.e., resilience) gradually improves in each generation, until finally the target resilience is reached.

7.2.4 Resilience-Enhancing Compiler Optimizations

In this paper, we restricted ourselves to finding compiler optimizations that preserve the error resilience of the unoptimized 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.

To explore this notion, we modified our GA-based algorithm to an Unbounded GA-based algorithm. The Unbounded GA-based approach is the same as our GA-based approach, except that 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 7.5. As can be seen, the candidate solutions obtained from the Unbounded GA-based approach generates optimization sequences with resilience either comparable to or better than the GA-based approach. However, the difference in the geometric 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 GA-based approach achieves similar results as the Unbounded GA-based approach, but takes much less time.

Figure 7.6 shows the mean fitness score of the population during the lifetime of the Unbounded GA-based approach as it progresses from one generation to another for each of the benchmarks. We normalize the number of generations and the

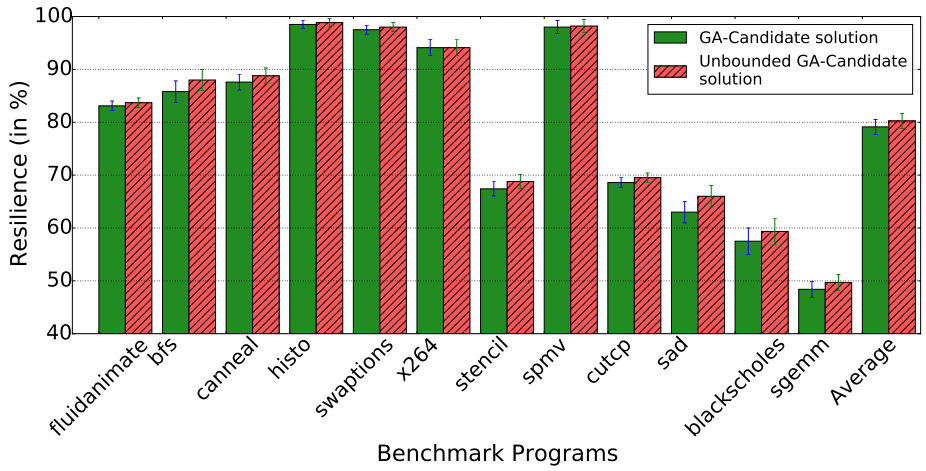

Figure 7.5: Resilience of the candidate solutions obtained from GA-based and Unbounded GA-based approaches.

fitness score to the total number of generations and the average fitness score of the last generation's population for the benchmark. As can be seen from the figure, the average fitness score of the population improves over generations, and remains constant after a particular generation. This is because either the algorithm has gotten stuck in the local maxima or it has attained the maximum resilience that can be obtained from the given set of individual optimizations in the initial population.

7.3 Limitations of our Approaches

While our approaches lead to faster convergence than the random walk, they have three limitations. First, as with all optimization techniques, the GA-based approach can get caught in local maxima and result in sub-optimal solutions. This is partially mitigated by the choice of the parameters such as the mutation rate, but ultimately, there is no guarantee that the approach will converge to the optimal solution. However our SA-based approach would probably prevent from getting stuck at local maxima. A second limitation is that our approaches can take a long time to converge to a solution. Although the algorithms itself are very fast, the fault injection to evaluate the resilience of the candidates in each generation takes a long time. We can however parallelize the injections as there are no dependencies among individ-

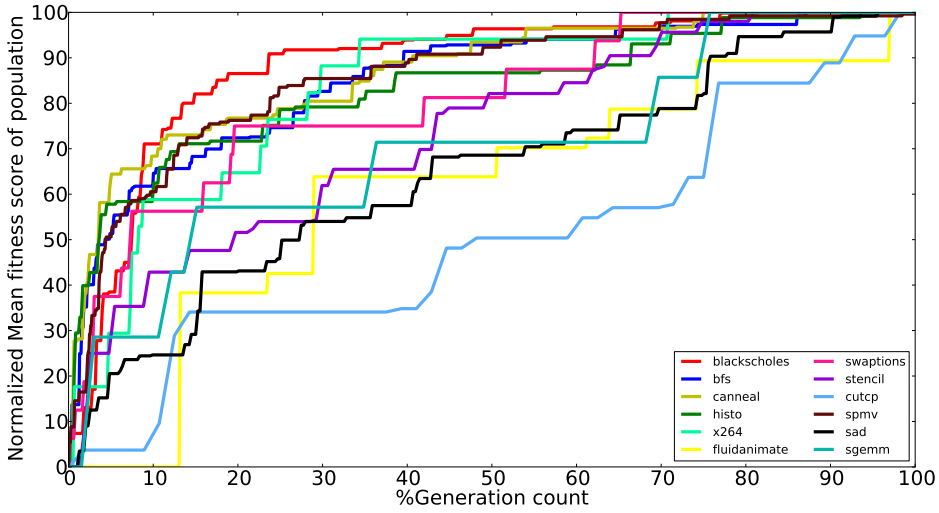

Figure 7.6: Mean population fitness scores during the Unbounded GA evolution process for each program.

ual injection runs, to obtain faster running times, at the cost of higher computational resources. A third limitation of our approaches is that our results were obtained on our x86-based hardware platform, and may not apply to other platforms. To mitigate this issue, we chose only the optimizations that are machine-independent, and are hence portable across platforms. Finally, we have considered only SDC's in our definition of resilience as they are the most critical outcomes, however crashes and hangs may also be important in some cases. These can be considered by suitably modifying the fitness function and score calculation.

7.4 Summary

In this chapter, we performed various analysis on the results. We first analyzed the results of our GA-based and SA-based approaches, to chose the most suitable approach. Based on this comparison we recommended the GA-based approach over SA-based approach and further examined why our approach performs better in this space. We finally discussed the limitations of both the approaches.

Chapter 8

Related Work

8.1 Effect of Compiler Optimizations on Resilience

Demertzi et al. [8] have analyzed the effect of standard optimization levels on the application's vulnerability. Unlike us, they do not consider the final outcome of the application due to the error, and whether the error results in an SDC. Further, our technique uses fault injections to evaluate the applications' resilience, while they use ACE analysis [21], which has been shown to be less accurate than fault injection [36].

Rehman et. al. [26] devise new optimizations that attempt to decrease the overall vulnerability of the program. The main difference between our work and theirs is that we consider the standard compiler optimizations rather than devise new optimizations for resilience. Further, our technique is able to increase the resilience of the application, without compromising on its performance.

Sangchoolie et. al. [28] consider the effect of compiler optimization level on SDCs for different applications. The main difference between their work and ours is that we consider individual optimizations such as *loop invariant code motion*, rather than the aggregate optimization level such as O1. This is important for two reasons. First, the optimization levels such as O1 and O2 are developed for performance and/or memory reasons, and do not consider error resilience. As a result, when choosing a standard optimization level, one may be including optimizations that hurt error resilience, leading to sub-optimal choices. Secondly, and

more importantly, the resilience effect of optimization levels may vary for different hardware platforms and applications. Therefore, it is important to develop an automated technique for choosing the appropriate set of optimizations for a given hardware platform and application.

Thomas et al [34] 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. Based on this study, they recommend a set of optimization that are "safe" against EDCs, i.e., do not significantly increase the EDC rate. 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. Such heuristics may not work well for all platforms and applications. Cong et al [5] proposed a metric for measuring the loop reliability to analyze the impact of loop transformations on software reliability. Similar to the previous work they mainly focus on soft computing applications and EDC outcomes. Also they restrict their analysis for loop optimizations, while we consider all compiler optimizations in general.

8.2 Choosing Compiler Optimizations

There has been a substantial amount of work to automatically choose the best set of compiler optimizations for performance improvement or memory reduction, for a given platform and application [4, 23]. Researchers have proposed the use of various search heuristics algorithms for this purpose (e.g., simulated annealing [38]). The most popular techniques in this space has been genetic algorithms (GAs) and Simulated Annealing (SA), which had been used to find compiler optimization sequences that optimize for performance [16, 32, 38], energy consumption [30], and code size [6]. None of them consider error resilience, which is our focus.

8.3 Software Errors and Genetic Algorithms

There has been considerable work on the use of genetic algorithms to repair software errors in programs [15, 18]. Given a test suite with one or more failing test cases, the main idea is to repair the program by (1) localizing the line of code which is responsible for the failure, and (2) replacing the faulty line with another line from the same program in an adaptive and iterative fashion using GA, till all the test cases in the suite pass. There are two main differences between this work and ours. First, we target hardware faults, while they target software bugs. Secondly, we aim to make the program resilient without knowing ahead of time where it will fail. Thus, our approach does not require a failing test case or test suite to guide the adaptation.

Chapter 9

Conclusion and Future Work

Compiler optimizations perform code transformations for improving the performance of an application. They mostly target to improve a specific aspect of the program like time, memory or code size. However, these code transformations would mostly affect the error resilience of the program.

In this work, we first performed a fault injection study to analyze the effect of individual optimizations on program's resilience. Based on the study we observed that some optimizations degrade program's resilience and others improve it. We also observed that this effect of optimizations on program's resilience depends on the application's characteristics as well. Hence we proposed automated techniques for finding resilience friendly compiler optimizations for a given application.

We leveraged search heuristic algorithms, Genetic Algorithms (GA) and Simulated Annealing (SA) to find an optimization sequence that improves the program's performance without degrading its error resilience. We evaluated our techniques on twelve benchmark programs, and found that it finds optimization sequences that provides resilience on par with or even better than the standard optimization levels (O1, O2 and O3), while achieving comparable performance improvement as the optimization levels. Further, it also lowers the vulnerability of applications, compared to the standard optimization levels.

For future work, we plan to look at aspects other than performance, such as code size, energy efficiency, and trade them off for error resilience. It would be interesting to explore different strategies to avoid getting stuck at local maxima

in our techniques. This might lead to obtaining better solutions. For example, modifying the selection strategy at the tournament selection step in the GA-based approach; adaptively changing the mutation and crossover rate over the generation in the algorithm; exploring other suitable operations for mutation, crossover and neighbor state; finding a hybrid algorithm based on our two approaches.

Further, resilience friendly compiler optimizations with similar impact on applications with similar behavior can be analyzed. The sequences obtained for these applications can be grouped together and fed as the population/input in our algorithm for obtaining better optimization sequences.

Based on the resilience-friendly sequences obtained from our approach, it would be interesting to find a model that can predict the effect of a give optimization on program's resilience. For this, the impact of the optimizations on application's code structure has to be studied.

Bibliography

[1] J. E. Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms and their applications*, 1985. → pages 20

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008. → pages 3, 23, 28, 31

[3] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *MICRO*, Nov. 2005. ISSN 0272-1732. doi:10.1109/MM.2005.110. URL http://dx.doi.org/10.1109/MM.2005.110. → pages 1

[4] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO*, March 2007. doi:10.1109/CGO.2007.32. → pages 58

[5] J. Cong and C. H. Yu. Impact of loop transformations on software reliability. In *ICCAD*, 2015. → pages 58

[6] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *LCTES*, 1999. doi:10.1145/314403.314414. URL http://doi.acm.org/10.1145/314403.314414. → pages 2, 58

[7] 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. → pages 20

[8] M. Demertzi, M. Annavaram, and M. Hall. Analyzing the effects of compiler optimizations on application reliability. In *IISWC*, 2011. → pages 1, 2, 11, 57

[9] G. E. Dieter and D. Bacon. *Mechanical metallurgy*, volume 3. McGraw-Hill New York, 1986. → pages 7

[10] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *SIGARCH*, 2010. → pages 1

[11] J. Grefenstette. Optimization of control parameters for genetic algorithms. *SMC*, Jan 1986. ISSN 0018-9472. doi:10.1109/TSMC.1986.289288. → pages 6

[12] S. K. S. Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi. GangES: Gang error simulation for hardware resiliency evaluation. In *ISCA*, 2014. → pages 29

[13] R. L. Haupt. Optimum population size and mutation rate for a simple real genetic algorithm that optimizes array factors. In *APSURSI*, 2000. → pages 32

[14] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. 1992. ISBN 0262082136. → pages 6

[15] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486893. → pages 59

[16] S. R. Ladd. Analysis of compiler optimizations via an evolutionary algorithm. https://packages.debian.org/sid/devel/acovea. → pages 2, 58

[17] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004. → pages 3, 13, 23, 30

[18] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38(1):54–72, Jan. 2012. ISSN 0098-5589. doi:10.1109/TSE.2011.104. URL http://dx.doi.org/10.1109/TSE.2011.104. → pages 59

[19] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Swat: An error resilient system. *SELSE*, 2008. → pages 1

[20] A. Lim, B. Rodrigues, and X. Zhang. A simulated annealing and hill-climbing algorithm for the traveling tournament problem. *European Journal of Operational Research*, 174(3):1459–1478, 2006. → pages 8

[21] 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. → pages 6, 57

[22] 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. ISSN 0885-8950. doi:10.1109/59.207317. → pages 20

[23] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO*, March 2006. doi:10.1109/CGO.2006.38. → pages 58

[24] B. J. Park, H. R. Choi, and H. S. Kim. A hybrid genetic algorithm for the job shop scheduling problems. *Computers & industrial engg.*, 2003. → pages 20

[25] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer. Application-based metrics for strategic placement of detectors. In *PRDC*, Dec 2005. doi:10.1109/PRDC.2005.19. → pages 27, 28

[26] 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. → pages 57

[27] M. Richmond. Examples of uncertainty calculations. http://spiff.rit.edu/classes/phys273/uncert/uncert.html. → pages 45

[28] 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. → pages 1, 2, 57

[29] 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. ISBN 1-55860-006-3. URL http://dl.acm.org/citation.cfm?id=93126.93145. → pages 6

[30] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. Post-compiler software optimization for reducing energy. *SIGARCH*, Feb. 2014. doi:10.1145/2654822.2541980. URL http://doi.acm.org/10.1145/2654822.2541980. → pages 58

[31] M. Srinivas and L. Patnaik. Genetic algorithms: a survey. *Computer*, 1994. ISSN 0018-9162. doi:10.1109/2.294849. → pages 20

[32] M. Stephenson, U.-M. O'Reilly, M. C. Martin, and S. Amarasinghe. Genetic programming applied to compiler heuristic optimization. In *EuroGP*, 2003. ISBN 3-540-00971-X. URL http://dl.acm.org/citation.cfm?id=1762668.1762691. → pages 58

[33] 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*, 2012. → pages 3, 31

[34] A. Thomas, J. Clapauch, and K. Pattabiraman. Effect of compiler optimizations on the error resilience of soft computing applications. In *AER*, 2013. → pages 1, 2, 11, 27, 28, 58

[35] P. van Laarhoven and E. Aarts. Simulated annealing. In *Simulated Annealing: Theory and Applications*, volume 37 of *Mathematics and Its Applications*, pages 7–15. Springer Netherlands, 1987. ISBN 978-90-481-8438-5. doi:10.1007/978-94-015-7744-1_2. URL http://dx.doi.org/10.1007/978-94-015-7744-1_2. → pages 7

[36] N. J. Wang, A. Mahesri, and S. J. Patel. Examining ACE analysis reliability estimates using fault-injection. In *SIGARCH*, 2007. → pages 57

[37] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *DSN*, 2014. → pages 30

[38] S. Zhong, Y. Shen, and F. Hao. Tuning compiler optimization options via simulated annealing. In *FITME*, Dec 2009. doi:10.1109/FITME.2009.81. → pages 2, 58