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

Nithya Narayananmurthy
Karthik Pattabiraman
Matei Ripeanu

University of British Columbia (UBC)
Motivation: Hardware Errors

• Soft errors are becoming more common in processors
  • Caused by alpha particles and cosmic rays
  • More sensitive to errors as feature sizes shrink

Source: Shekar Borkar (Intel) - Stanford talk
Traditional Solutions

• Dual Modular Redundancy (DMR) (e.g., IBM Mainframes, Tandem Non-Stop etc.)

• DMR incurs significant energy overheads - undesirable in commodity systems
Our Approach: Good Enough Dependability

Leverage the properties of the application to provide targeted protection, only for the errors that matter to it

Targeted protection mechanisms
Error Resilience

• Silent Data Corruption (SDC)
  • Incorrect output without any indication (Most critical)

• Error Resilience:
  • Conditional probability that a fault does not produce an SDC given that it has occurred

• Vulnerability:
  • Product of error resilience and execution time of program
  • Assume transient faults occur uniformly over time
Compiler Optimizations

• Typically used to make programs faster by transforming the code
  • Processor performance has plateaued for a decade
This Paper

Compiler Optimizations

- Improve Performance!
- Resilience

Goal: Identify optimizations that provide same error resilience

Performance - Resilience trade off
RQ1: What effect do compiler optimizations have on a program’s resilience?
Fault Injection Study

- **Fault Model:** Faults that occur in the computational components and register files of the processor

- Single bit flip fault

- One fault per run

- Injected using LLFI fault injector
Fault Injection Study

- 10 random compiler optimizations from LLVM compiler
- Benchmark programs – Blackscholes and Swaptions (Parsec)
- 3000 fault injections
- Error bars: 1.8%, 0.7%
- Resilience normalized to the resilience of unoptimized program

Same optimization can have different effects for different programs
Example optimization - 1: Loop Invariant Code Motion (LICM)

<table>
<thead>
<tr>
<th>Un-optimized</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  for(i = 0; i &lt; 10; i++) 1  alpha = (x * c) + s;</td>
<td></td>
</tr>
<tr>
<td>2  { 2  for(i = 0; i &lt; 10; i++)</td>
<td></td>
</tr>
<tr>
<td>3     alpha = (x * c) + s; 3  {</td>
<td></td>
</tr>
<tr>
<td>4     rs1[i] = i + (alpha * 7); 4     rs1[i] = i + (alpha * 7);</td>
<td></td>
</tr>
<tr>
<td>5  } 5  }</td>
<td></td>
</tr>
</tbody>
</table>

LICM reduces overall resilience
Example Optimization - 2: LOOP-REDUCE

Un-optimized

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

Optimized

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   }

LOOP-REDUCE improves resilience
RQ1: Summary

• Different optimizations have different effects (degrade/improve/no impact) on error resilience

• Optimization’s effect on resilience differs based on the application characteristics – **no one size fits all**

**We need techniques for finding resilience-friendly optimization for a given application**
RQ2: Can we find resilience-friendly optimizations that preserve the error resilience of a given program?
Finding Resilience-Friendly Optimizations

• Impact of an optimization depends on:
  • Application
  • Hardware platform

• Error resilience is sensitive to the order of optimizations in a sequence

• Search space is very large \((2^n \times n!)\), where \(n\) is the number of optimizations

Meta-Heuristic Search Techniques
Genetic Algorithm (GA)

Evolve the solution by mutating it in every iteration till we converge to a candidate solution

- Kill the weak mutants based on fitness function
GA-based Approach

Application Source Code

Individual Optimizations

Initialization

Tournament Selection

Recombination Operations

Elimination

Termination

Candidate solution

One Iteration/Generation

Fitness Function

FF
GA-based Approach

Initialization

- instcombine
- licm
- Loop-unroll
- gvn
- inline
- sccp
- Loop-unswitch
- cse
- Loop-reduce

Population

Optimization

Fitness Score

- 57
- 59
- 50
- 59
- 57
- 61
- 64
- 52
- 55
GA-based Approach

ReCombination Operations

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Fitness Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>licm</td>
<td>57</td>
</tr>
<tr>
<td>instcombine</td>
<td>59</td>
</tr>
<tr>
<td>cse</td>
<td>50</td>
</tr>
<tr>
<td>inline</td>
<td>59</td>
</tr>
<tr>
<td>gvn</td>
<td>57</td>
</tr>
<tr>
<td>Loop-unswitch</td>
<td>61</td>
</tr>
<tr>
<td>Loop-unroll</td>
<td>64</td>
</tr>
<tr>
<td>sccp</td>
<td>52</td>
</tr>
<tr>
<td>Loop-reduce</td>
<td>55</td>
</tr>
</tbody>
</table>

Crossover - Append
- loop-reduce - loop-unroll 60
GA-based Approach

Recombination Operations

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Fitness Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>licm</td>
<td>57</td>
</tr>
<tr>
<td>instcombine</td>
<td>59</td>
</tr>
<tr>
<td>cse</td>
<td>50</td>
</tr>
<tr>
<td>inline</td>
<td>59</td>
</tr>
<tr>
<td>gvn</td>
<td>57</td>
</tr>
<tr>
<td>Loop-unswitch</td>
<td>61</td>
</tr>
<tr>
<td><strong>Loop-unroll</strong></td>
<td><strong>64</strong></td>
</tr>
<tr>
<td>sccp</td>
<td>52</td>
</tr>
<tr>
<td>Loop-reduce</td>
<td>55</td>
</tr>
</tbody>
</table>

Crossover - Append

- loop-reduce  - loop-unroll  60
Experimental Setup

• Twelve Benchmark programs
  • Five from PARSEC
  • Seven from Parboil

• LLVM compiler (widely used in industry)

• Performed fault injections using LLFI [DSN’14]
  • 1000 fault injections, one fault per run
  • Error bars: 0.85% - 2.5%
  • Total fault injections: 60,000 fault injections
Evaluation

• Resilience

• Execution Time

• Vulnerability

Compared with standard levels O1, O2 and O3 as most prior work focuses on the standard levels [Sangchoolie’14][Demertzi’10]
Results: Resilience

+ ve values: higher resilience (better)
- ve values: lower resilience (worse)

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Resilience Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>-4.64%</td>
</tr>
<tr>
<td>O2</td>
<td>-4.21%</td>
</tr>
<tr>
<td>O3</td>
<td>-3.80%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Resilience Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>GA</td>
<td>+4.46%</td>
</tr>
</tbody>
</table>
Results: Normalized SDC Rates

+ ve values: higher SDC rate (worse)
− ve values: lower SDC rates (better)

<table>
<thead>
<tr>
<th>Optimization</th>
<th>SDC rate Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>+26.50%</td>
</tr>
<tr>
<td>O2</td>
<td>+22.79%</td>
</tr>
<tr>
<td>O3</td>
<td>+23.38%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimization</th>
<th>SDC rate Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>GA</td>
<td>-23.99%</td>
</tr>
</tbody>
</table>
Results: Execution Time

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Performance Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>6.56%</td>
</tr>
<tr>
<td>O2</td>
<td>7.21%</td>
</tr>
<tr>
<td>O3</td>
<td>11.10%</td>
</tr>
<tr>
<td>GA</td>
<td>8.99%</td>
</tr>
</tbody>
</table>
Results: Vulnerability

Vulnerability = SDC rate * run time

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Vulnerability Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>GA</td>
<td>-33.46%</td>
</tr>
<tr>
<td>O3</td>
<td>-6.26%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Vulnerability Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>+7%</td>
</tr>
<tr>
<td>O2</td>
<td>+0.35%</td>
</tr>
</tbody>
</table>
RQ2: Summary

• Candidate solutions using Genetic Algorithms
  • Outperform optimization levels in resilience
  • Provide reasonable performance improvements better than O1, O2 and only slightly worse than O3 (by about 2%)

• Significant vulnerability reduction (by 27%) compared to O3, and much better than O1 and O2

• Takeaway: It is possible to achieve both high performance and high resilience using optimizations
  • But they must be carefully chosen on a per-application basis
Related Work

• Analyzed the impact of standard levels O1, O2 and O3 on program’s resilience[1 - Demertzi][2 - Sangchoolie]

• Proposed new optimizations that targets vulnerability reduction, without performance improvement [3 - Rehman]

• Focused on soft computing applications for EDC (Egregious Data Corruptions) outcomes [4 - Thomas][5 - Cong]

Conclusion

• Studied the impact of individual compiler optimizations on error resilience
  • Observed varied effects of optimizations on different programs

• Used Genetic Algorithms (GAs) to find resilience friendly compiler optimizations for each program

• Candidate solutions have much better resilience and lower vulnerability than standard optimizations levels with only small performance degradation