Efficient Soft Error Protection for Commodity Embedded Microprocessors using Profile Information

By: Daya Shanker Khudia, Griffin Wright, and Scott Mahlke
Presented By: Nicholas Kroetsch, Gabriella Rodriguez, Charles Light

Soft Errors

Soft Errors: transient faults caused by high energy particle strikes

- The amount of charge released by high energy particle strikes determines whether a transistor will malfunction
- Smaller transistors make processors more susceptible to soft errors
- When a transient fault occurs in a computer system it can corrupt the application output or crash the system
  - Data center crash
  - Shutdowns in an automotive factory

Focuses in Soft Error Research

- High-performance server market
- Fault protection via redundant multithreading
- Hardware checkers

Focus of this paper:

- Embedded system coverage through software based fault detection and instruction duplication (via profiling)
  - Lenient reliability requirements
  - No extra hardware

Contributions

1) A **software solution** which does not need any user annotations in the application to generate reliability-aware code and works on applications written in a variety of languages

2) A selective instruction duplication approach that leverages **memory profiling and edge profiling** in compiler analysis to identify and replicate a small subset of vulnerable instructions not covered by symptom-based fault detection.

3) **Novel use of value profiling f**or the generation of software symptoms.

4) Microarchitectural **fault injection experiments** to demonstrate the effectiveness of our proposed solution in terms of fault coverage and performance overhead.

Background and Motivation

Soft Error Rate (SER)

- 75-92% of transient faults get masked due application level masking
  - Total masking rate around 78% from all sources
- Memory cells are more vulnerable to soft errors because they use smaller transistors to achieve higher densities
- SER for combinational logic is expected to increase over time
- Voltage scaling worsens the problem of soft errors as smaller disturbances in circuits will be able to flip a bit

Instruction Duplication

Previously Proposed Duplication Methods:

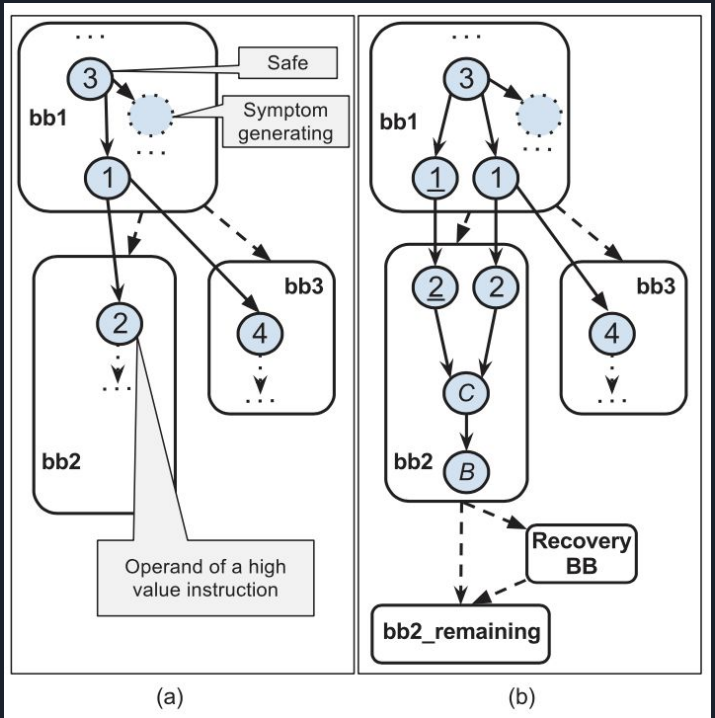
- *SWIFT*: recursively duplicates instructions by walking the data flow chains of the operands of stores and by protecting the control flow

- *Shoestring*: considers only global stores and by protecting the control flow only for the immediate branch that affects the execution of a global store

Classes of instructions proposed in this paper:

- *Symptom-generating*: produce detectable symptoms if they consume a corrupted input
- *High-value:* corrupt the output of the program if they consume a corrupted input
- *Safe:* naturally covered by symptom-generating consumers

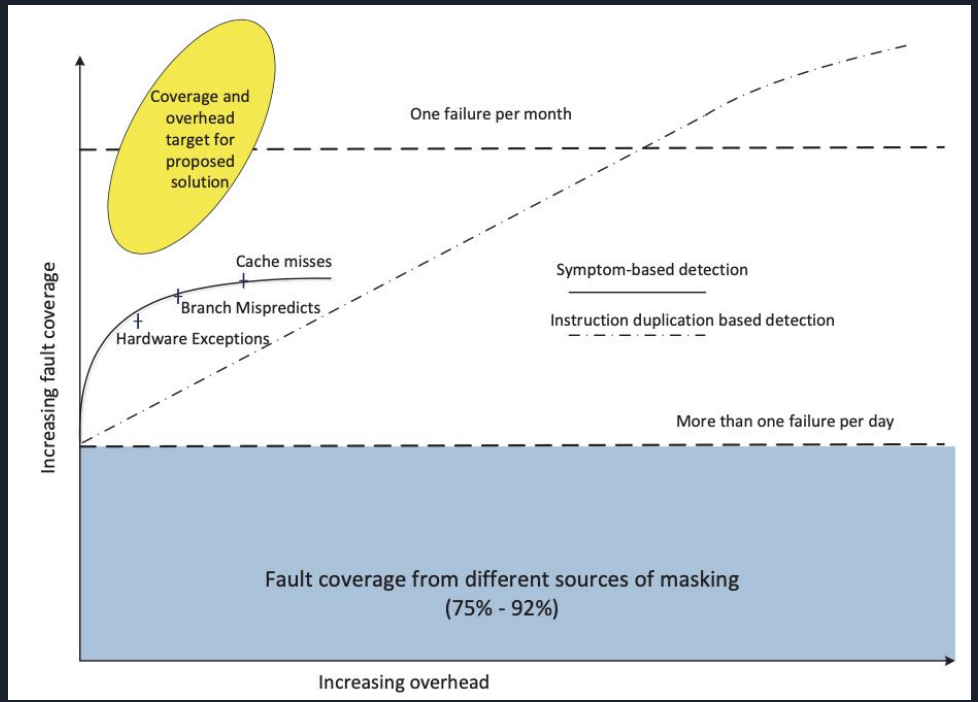
Instruction Duplication Example



(A) Original code
(B) Code after duplication

- Node 3 is classified as safe because it has a symptom generating consumer
- The duplication of nodes 1 and 2
- The result of the high value node (2) requires a compare and branch to determine if the instruction is executed normally

Proposed Solution Landscape



This paper proposes combining symptom based detection and instruction duplication based detection to reach their coverage and overhead target (emphasized in yellow)

Profile Based Duplication

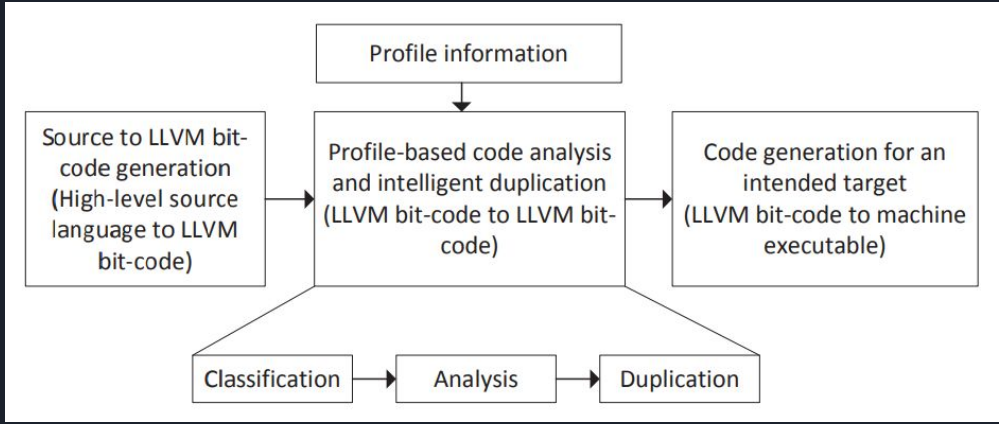
- Profiling information has been successfully used in profile-guided optimizations to improve performance

- This paper is the first time in the context of ‘code duplication for protection against soft errors’ that the following profiling methods are used:
  - Edge profiling
  - Memory profiling
  - Value profiling

Proposed Solution

Proposed Solution

- Compile-time instruction duplication as an LLVM compiler pass
- Increased tolerance of soft errors, but <100% coverage by design
- Leverage application profile information to reduce overhead



High-Value Instructions

- Key idea: Applications mostly communicate to the outside world through I/O library calls
- Protect these instructions by duplicating their producer chains:
  - Library calls
  - Function calls
- Other programs use memory mapped I/O
  - Can consider stores as high value to protect them
- Otherwise, use LAMP to profile memory data dependencies
  - Walk the producer chain of high-value instructions
  - If a load is encountered then consider all stores that could alias to be high value instructions and duplicate their producer chains

Profile-based Overhead Reduction

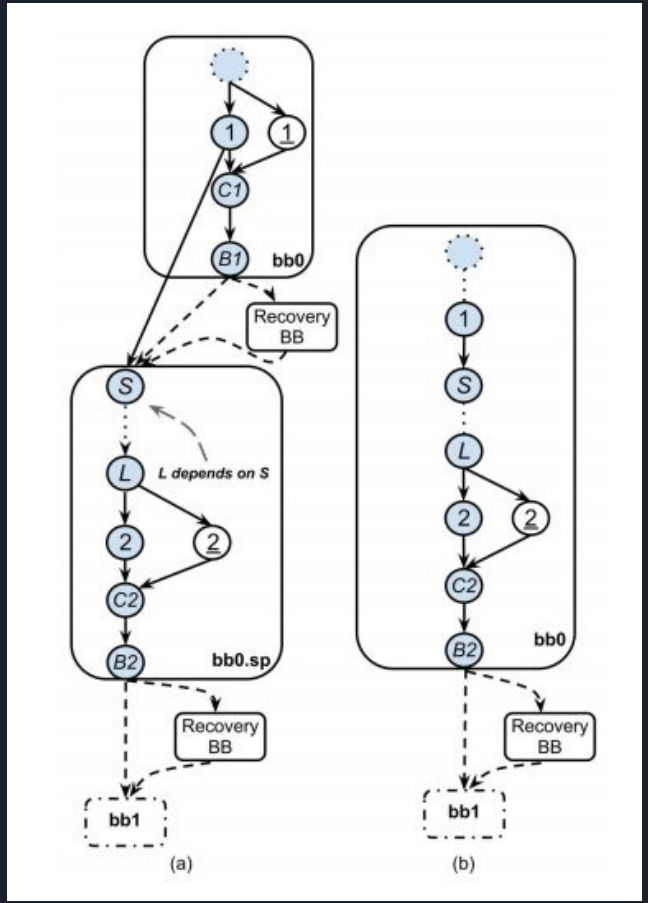
Key contribution of this work is using profile information to reduce overhead while minimizing coverage loss

- Edge Profile based pruning:
  - A frequently executed instruction should not be duplicated to protect an infrequently executed one
    - Ex: a loop that executes 1000x, with a conditional inside that will call a library function once.
  - Probability of a soft error affecting an infrequently executed instruction is low

Silent Store Optimization

Silent Store: a store that writes the same value to a memory location that was already present in that location

- Up to 72% of total stores are silent in some benchmarks
- Based on profile data, stores that are almost (>80%) silent will stop recursive duplication
  - Even if a corrupted value is written, it will be written correctly by the next execution of the store



Software Symptom Generation using Value Profiling

- New method to generate software symptoms based on profiling
  - If an instruction generates the same result nearly 100% of the time, compare the value against the one generated at runtime
- If the value generated differs, assume a fault occurred and trigger recovery
  - Only requires insertion of a single compare and branch instruction rather than duplication, lower overhead than duplicating the full producer chain for the instruction
  - If comparison fails twice from the same place after error recovery, then the different value is considered correct and an error is not triggered

Experimental Setup

Experimental Setup

- 10 applications from SPECINT2000 benchmark suite compiled using the modified LLVM compiler pass
  - Single-threaded applications
- Simulated in-order ARMv7 core with a single bit-flip model
  - Supports the ability to roll back processor state to a clean checkpoint within a window of 1000 instructions

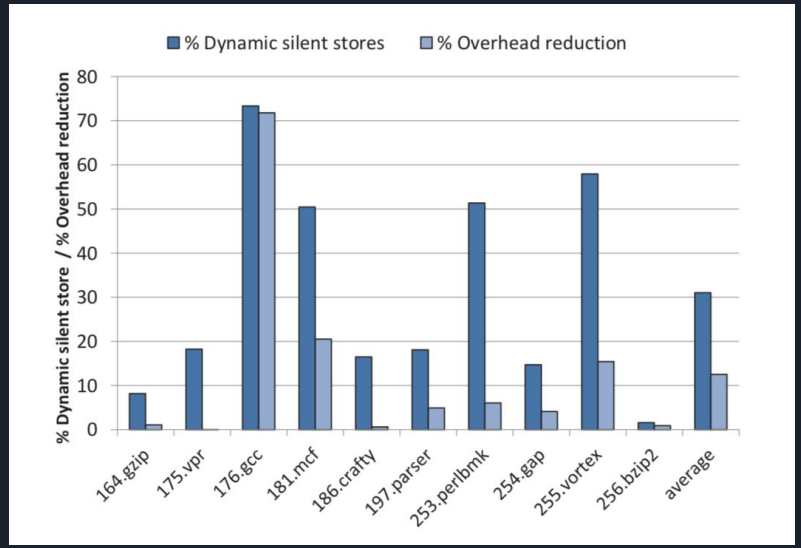
Fault Injection Framework

- Single bit-flip fault model
- Faults injected randomly into the register file
  - Faults in other structures would manifest in the register file
  - 100 fault injection trials per benchmark
- 4 categories of result:
  - Masked: no visible effect
  - Covered by symptom-based detection
  - SWDetect: Detected by instruction duplication
  - Silent corruption or infinite loop

Experimental Results

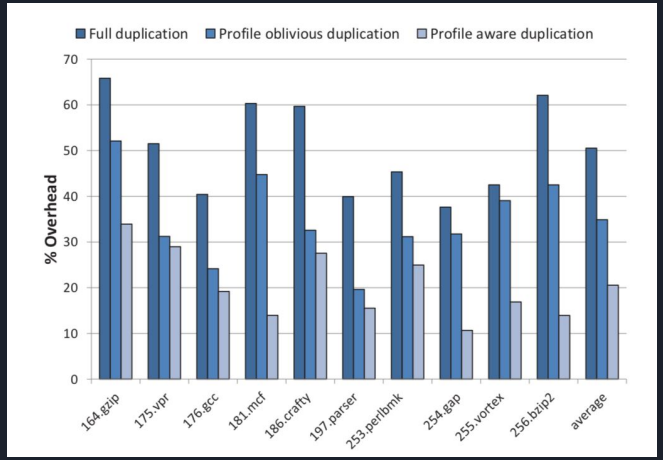
Utilizing Silent Stores

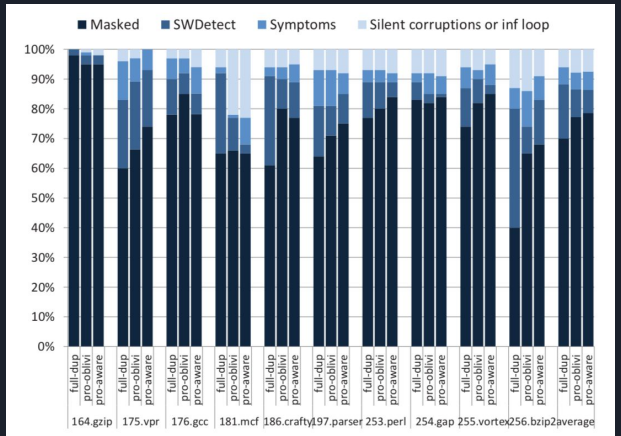
- Any memory write instruction that writes the same value over 80% of the time is considered a silent store, and does not need to be fully duplicated for optimization
- More than 50% of stores in 4/10 benchmarks were these, although percentage in others was low
- Overhead reduction significantly associated with prevalence of these (more than 70% in one benchmark but far lower in the others).



Performance Overhead, Fault Coverage

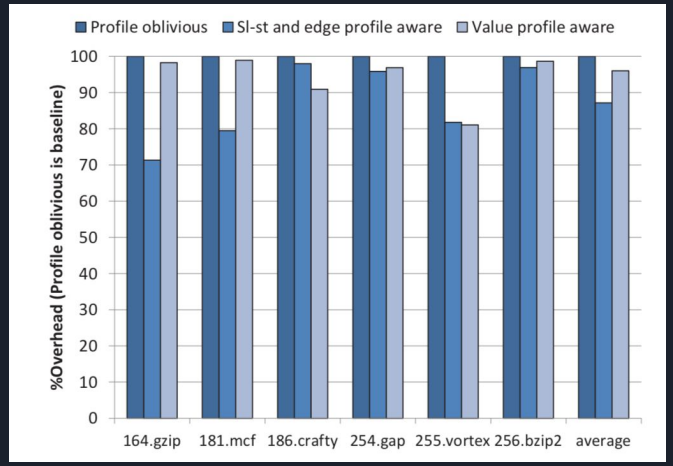
- Full duplication has about 50% overhead, 94% coverage
- Profile-oblivious method (baseline from Shoestring's prior work where duplication terminates with safe instructions) has about 35% overhead, 92% coverage
- Profile-aware method has about 20% overhead, 92% coverage

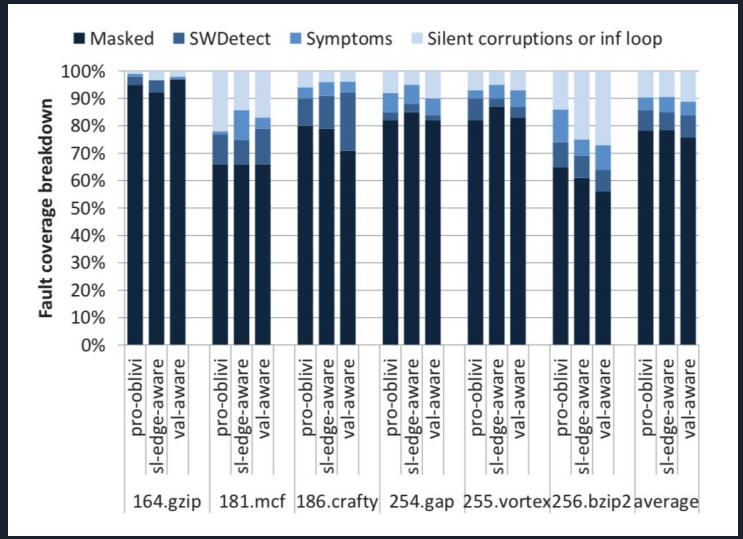




Effects of Each Technique

- Combining silent store optimization with edge profile-aware duplication reduces overhead on average by 12.78%.
- Value profiling reduces overhead by 5.9%, slightly increases number of faults covered





Conclusions

- Dealing with soft errors is a major concern to make transistor sizes at or below 16 nm feasible to work with.
- These techniques in the paper do work (achieve both high fault coverage and low overhead)
- Other implications: not ISA-dependent, can achieve both high detection and low overhead in a software solution.
- Future work could involve making further optimizations or validating the concept for other fault injection sites, multithreaded programs, or out-of-order processors.