Software approaches for *in-time* Resilience

Aviral Shrivastava, Moslem Didehban
Arizona State University, Cadence Design Systems

CML

Soft and hard errors

- Reliability is one of the most important design concerns

- Main sources of hardware unreliability
  - Soft error, aka transient fault
  - Hard error, aka permanent fault

MTBF (One car perspective)	1 year	120 years
MTBF (Toyota perspective: 10 million cars sold last year)	3 seconds	~6 minutes

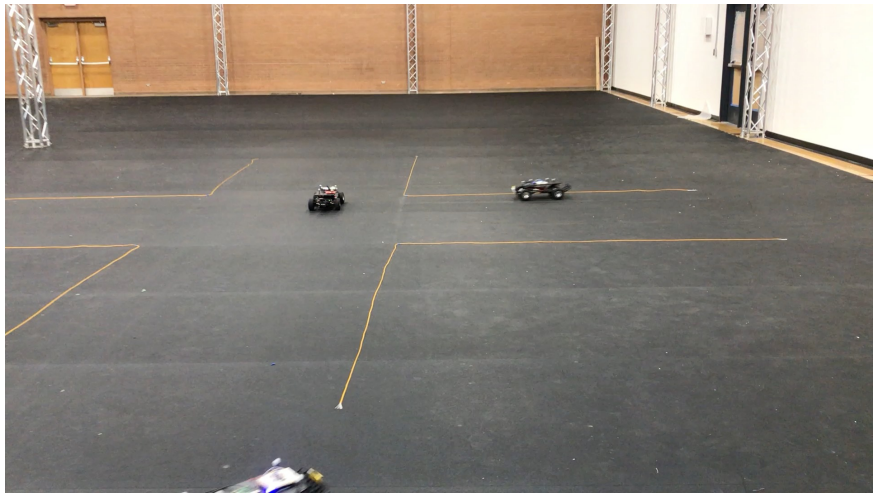
Why protect in software?

- Protection is flexible
  - Can be applied only to the critical applications or parts thereof
- No hardware modifications required
  - Can be applied on any past, present and future processor
  - Good/agnostic coverage of uncore components
- Can potentially be more efficient
  - Several strong error masking effects exist in hardware
  - If fault does not propagate to the software layer, we do not care
  - Cohesive with the concept of approximate computing

Need *in-time* Resilience

Microsoft Clippy

- Computing used in many safety-critical applications
  - No longer just word processing!
- In many of these applications, the computations have to meet strict timing constraints
  - Detection and recovery from error must be quick



Traffic intersection design for autonomous cars. Ref:

- Mohammad Khayatian, Mohammadreza Mehrabian and Aviral Shrivastava. RIM: Robust Intersection Management for Connected Autonomous Vehicles. In Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS), 2018.
- Edward Andert, Mohammad Khayatian and Aviral Shrivastava. Crossroads - A Time-Sensitive Autonomous Intersection Management Technique. In Proceedings of The 54th Annual Design Automation Conference (DAC), 2017.

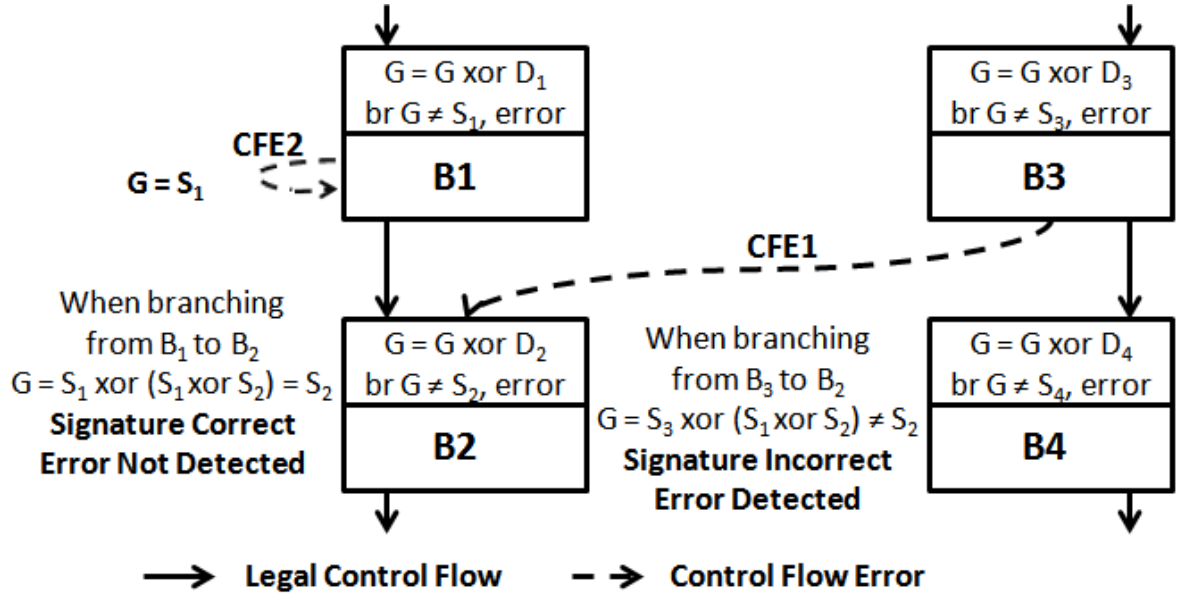
Related Work

- HPC error-resilience research
  - + Many software techniques
    - primarily based on checkpointing
  - - In-time resilience is not so important

- General computing
  - Redundancy
    - Our old faithful… but seemingly high overheads
  - Control Flow Checking
    - Intuition: If there is an error most probably the control flow the program will change [CFCSS, CEDA, ACCE, ECCA, YACCA…]
      - Control flow: sequence of execution of instructions
  - Symptom-based fault detection
    - Intuition: If there is an error, then it will create some symptoms [ReStore, SWAT, mSWAT…]
      - Symptoms, e.g., branch misprediction, TLB miss
  - Algorithm-based fault testing
    - Sort of checksums, but limited usage [ABFT]

Overall the concern with software techniques has been about their effectiveness

Control Flow Checking



Signature based CFC

- assign each BB a signature
- write it in a register
- check the register in the next BB

- **DAC 2014:** Quantitative analysis of Control Flow Checking Mechanisms for Soft Errors
  - CFC techniques can only detect about 4% of soft errors
  - 15-20% performance overhead

Symptom-based Checking

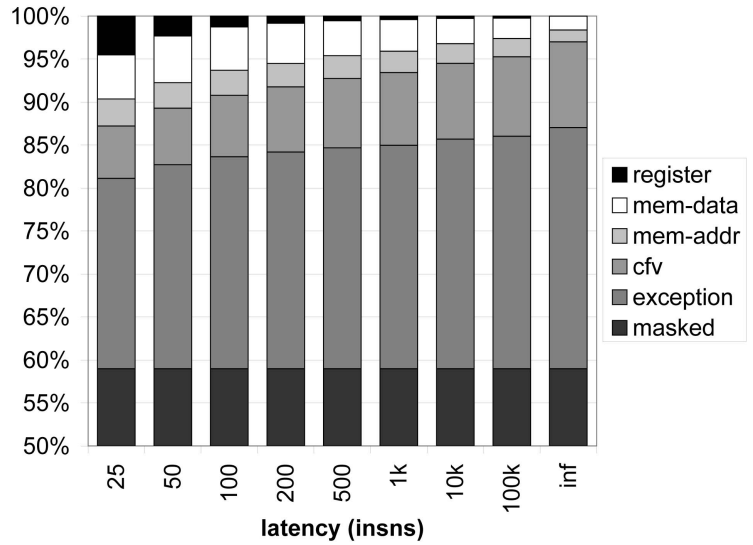


Fig. 2. Virtual machine fault injection.

Symptom-based error detection

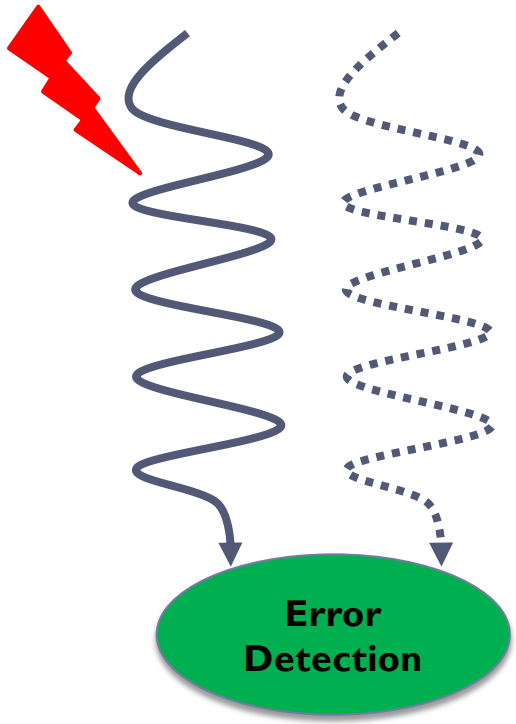
- Define some symptoms, e.g., BP miss
- Commit only when no symptom generated
- If symptom seen, re-execute
- Assume second execution is correct

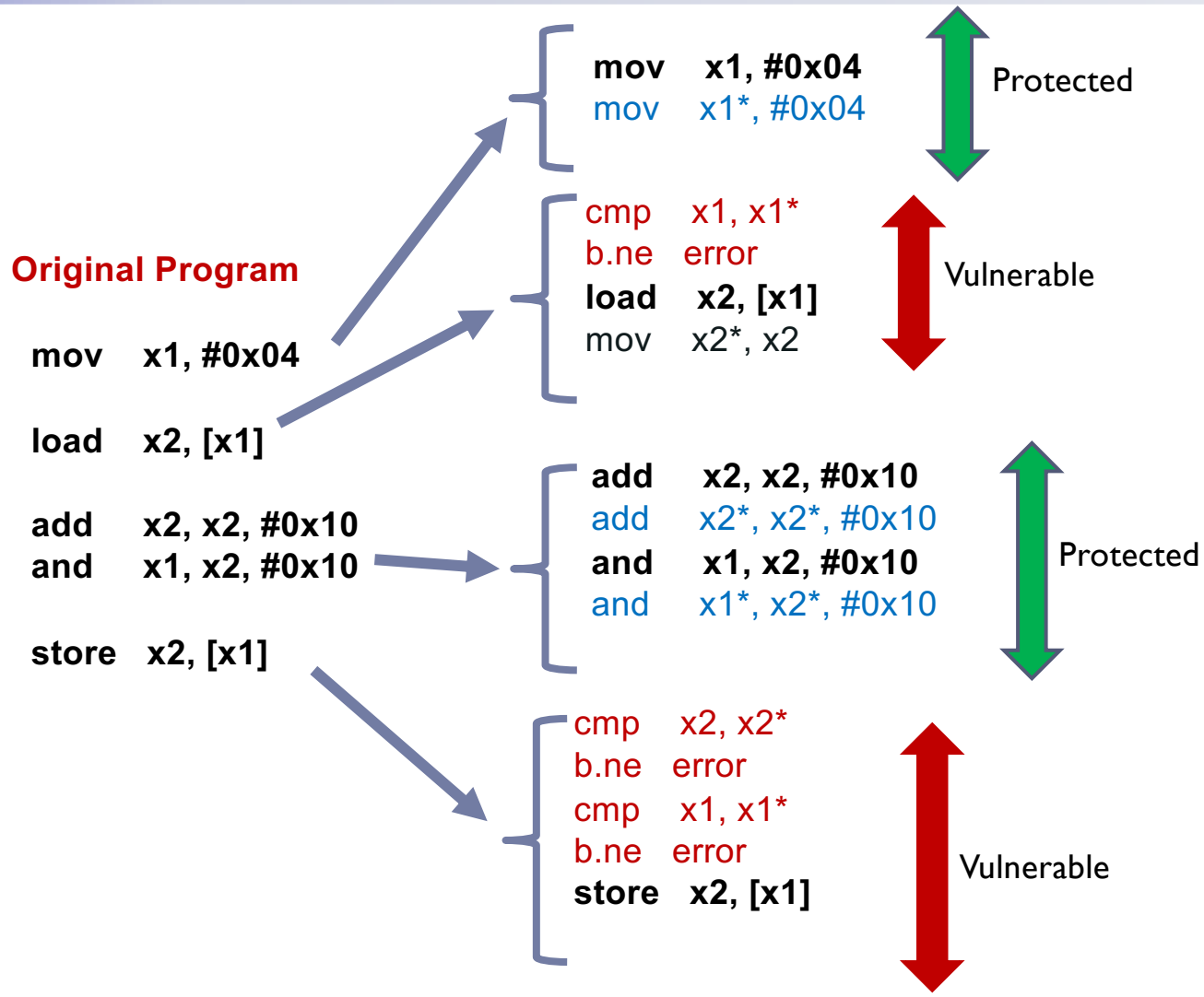
- Our experiments show
  - Coverage achieved is proportional to the amount of re-execution

Redundancy – No shortcut to protection

- What granularity?
  - Coarse-grain or fine-grain
- Coarse-grain: process-level, function-level etc…
  - You have to compare the whole memory.
  - High overhead, not in-time
- Fine-grain: Instruction-level
  - Can check one “store”
  - Lower overhead, and in-time detection

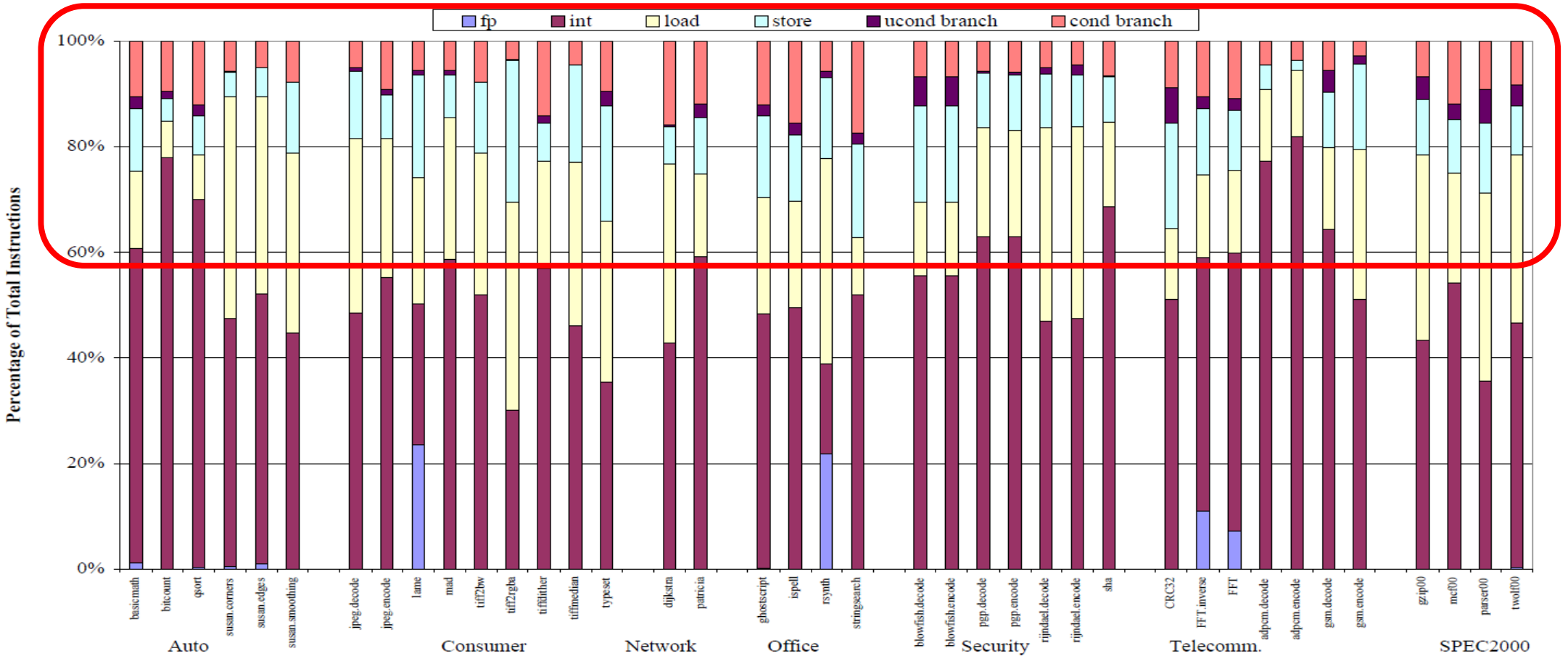
Error Detection by Instruction Replication





Examples: SWIFT[2005], Shoestring[2010], DRIFT[2013], SIMD-Based Soft Error Detection [16], IPAS [2016]

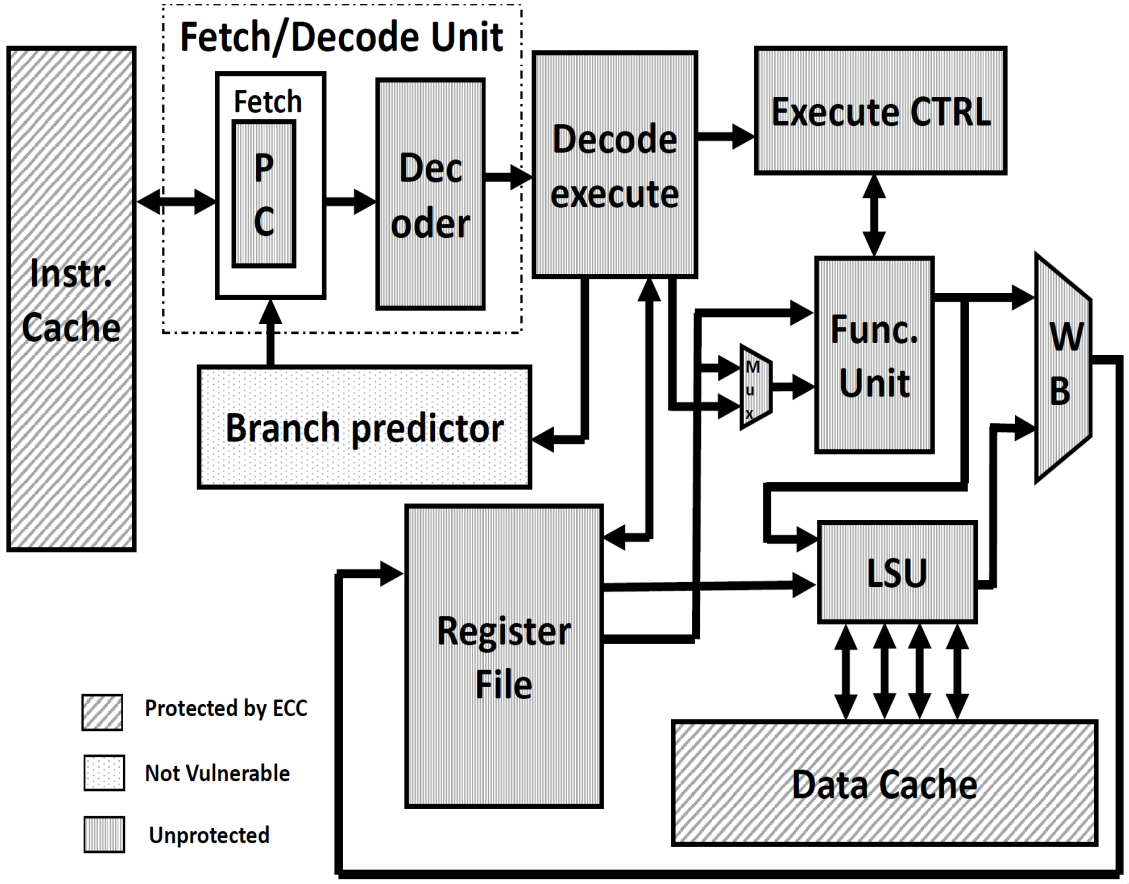
More than 40% of the instructions not duplicated!!



"MiBench: A free, commercially representative embedded benchmark suite." The University of Michigan.

Evaluation Set up

- OpenRISC architecture
  - Synthesizable Verilog Code of OR1k implementation



Component	Fault Model	# of fault sites
Register File	Single Event Upset[a]	1024
Fetch/Decode Unit	Single Event Upset	200
Decode/Execute Unit	Single Event Upset	216
Execute/Control Unit	Single Event Upset	183
Write/Back Unit	Single Event Upset	32
ALU	Single-event transient[b]	36
LSU	Single-event transient	101

[a] Errors are injected on flip-flops.
[b] Errors are injected on combinational circuits.

- Randomly pick a fault site and a cycle, and flip the value for one cycle by XORing that value with '1'.
- No micro benchmarking

CML

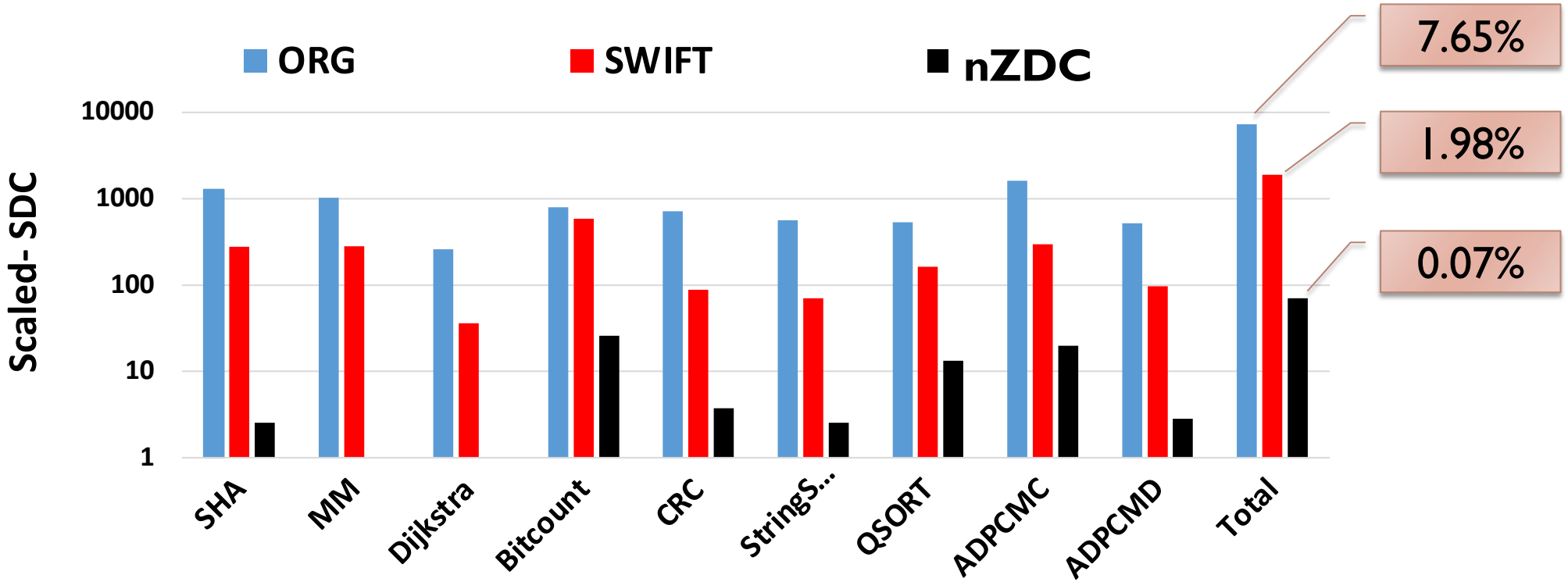
Fault Injection Results

- 10,600 FI experiments per each version of a program
- In total, about 1 mil RTL-level FI experiments

Reliability based on number of nines

Version	Coverage	Overhead
ORG	90%	1x
SWIFT	90%	2.7x
nZDC	99.9%	2.9x

Scaled-SDC = # of SDCs * runtime overhead



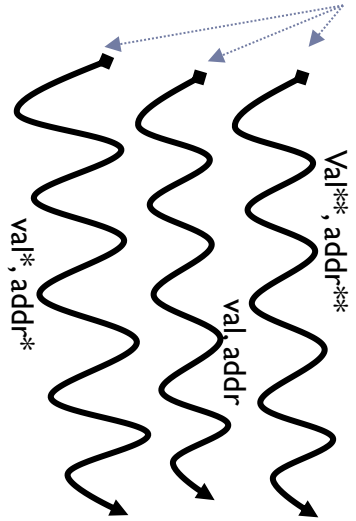
SWIFT: 3.8x SDC Reduction **nZDC:** 104x SDC Reduction

Web page: aviral.lab.asu.edu

CML

A Closer Look into SWIFT-R

Redundant computations



Majority-voter(val, val*, val)**

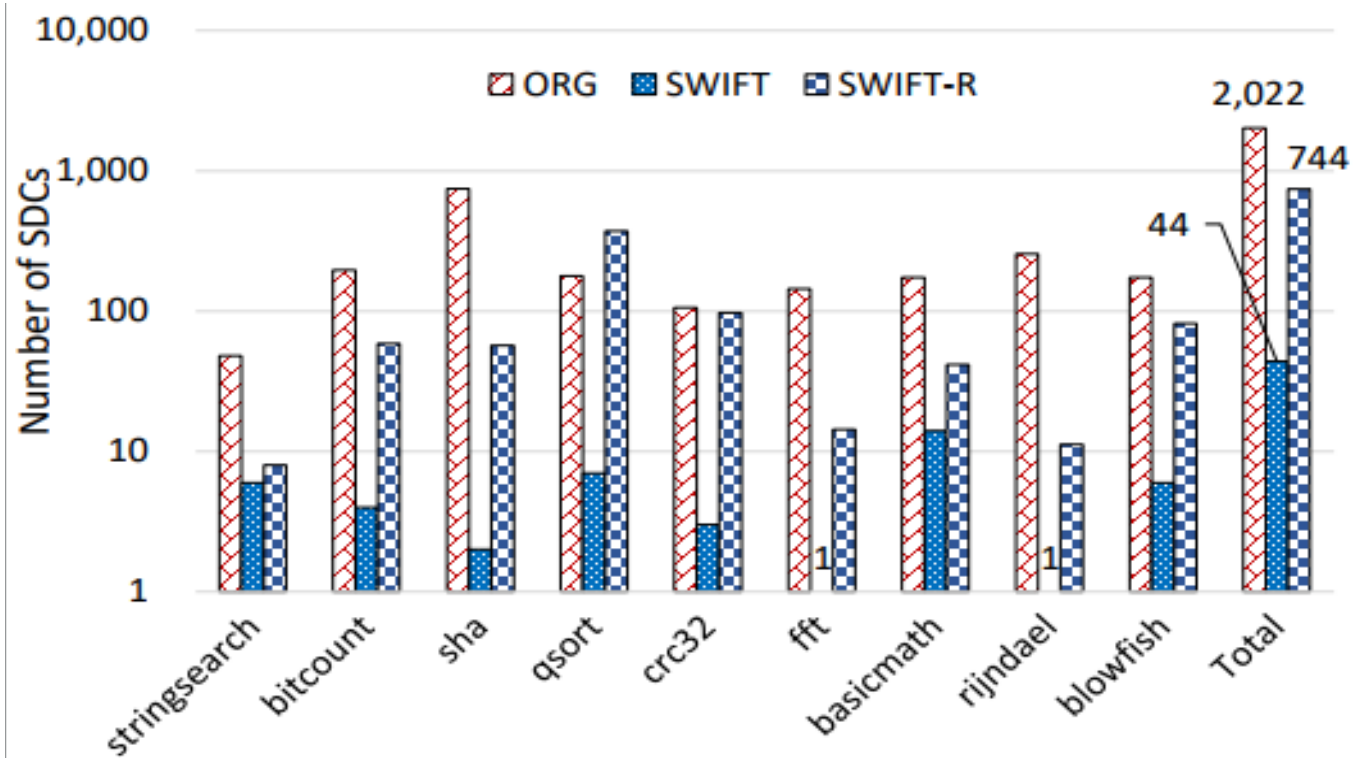
Majority-voter(adr, adr*, adr)**

store val→[addr]

```
if ((adr != adr*) || (addr != adr **) || (adr * != adr **)){
    if (adr == adr *)                  // addr ** is faulty
          adr ** = adr;
    else if (adr * == adr **)          // addr is faulty
          adr = adr *;
    else if (adr == adr **)            // addr * is faulty
          adr * = adr;
}
```

```
    movl   -4(%rbp), %eax
    cmpl   -8(%rbp), %eax
    jne    .L2
    movl   -4(%rbp), %eax
    cmpl   -12(%rbp), %eax
    jne    .L2
    movl   -8(%rbp), %eax
    cmpl   -12(%rbp), %eax
    je     .L6
.L2:
    movl   -4(%rbp), %eax
    cmpl   -8(%rbp), %eax
    jne    .L4
    movl   -4(%rbp), %eax
    movl   %eax, -12(%rbp)
    jmp    .L6
.L4:
    movl   -4(%rbp), %eax
    cmpl   -12(%rbp), %eax
    jne    .L5
    movl   -4(%rbp), %eax
    movl   %eax, -8(%rbp)
    jmp    .L6
.L5:
    movl   -8(%rbp), %eax
    cmpl   -12(%rbp), %eax
    jne    .L6
    movl   -12(%rbp), %eax
    movl   %eax, -4(%rbp)
.L6:
```

SWIFT-R causes 16x more SDC than SWIFT!!!!!



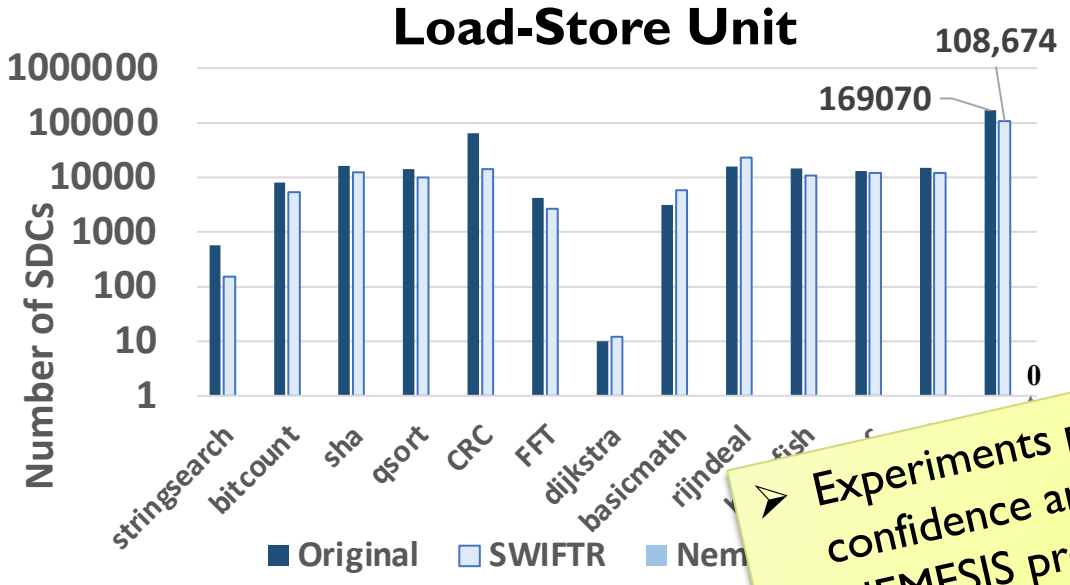
18K single bit fault injection experiments on register file.

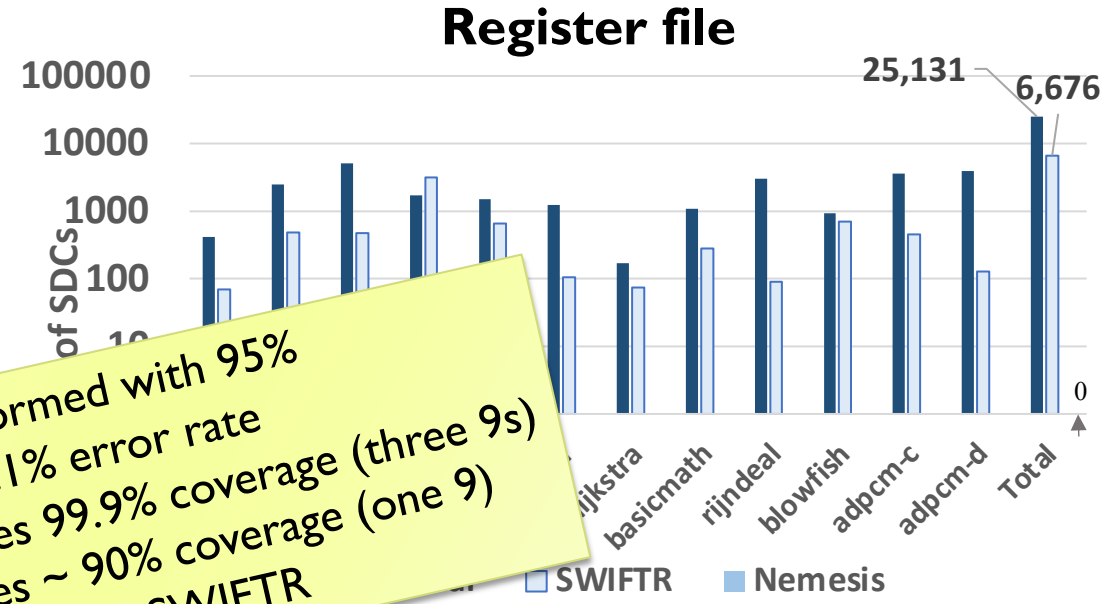
"**Organization of redundancy** and fault-tolerance for ultra-high reliability is a challenging problem: redundancy management can account for half the software in a flight control system and, **if less than perfect** can itself become the primary source of system failure." -- John Rushby [1995]

Why is SWIFT-R even worse than SWIFT?

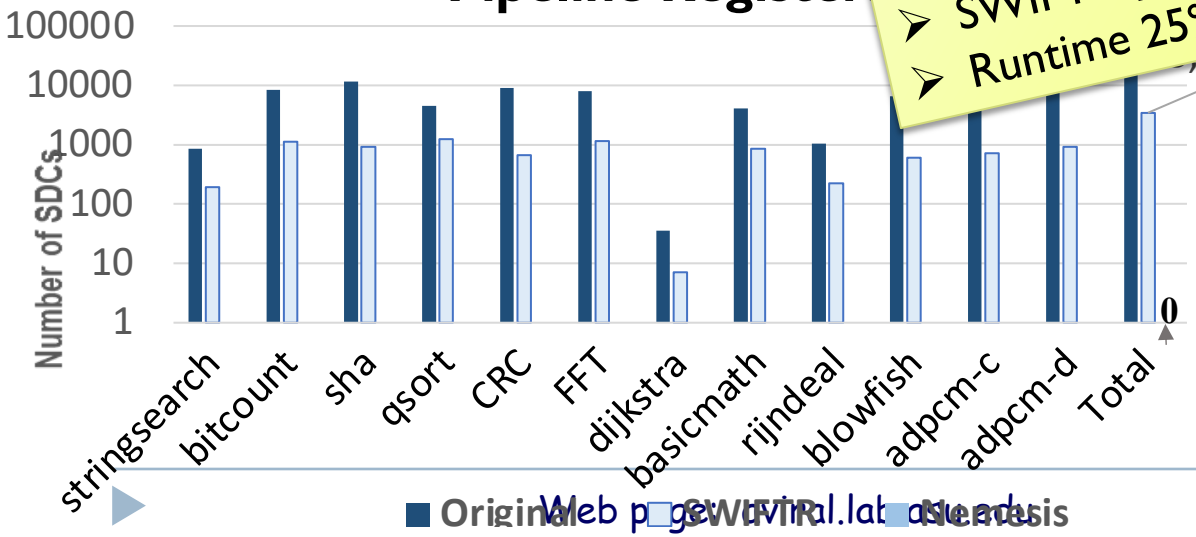
- More loads and stores in SWIFT-R, than in SWIFT
- Checker is much bigger in SWIFT-R, than SWIFT, and the checks in SWIFT and SWIFT-R are vulnerable
- If there is a fault in the checker, then SWIFT can detect errors later – in the next check, but SWIFT-R will mask the error symptoms

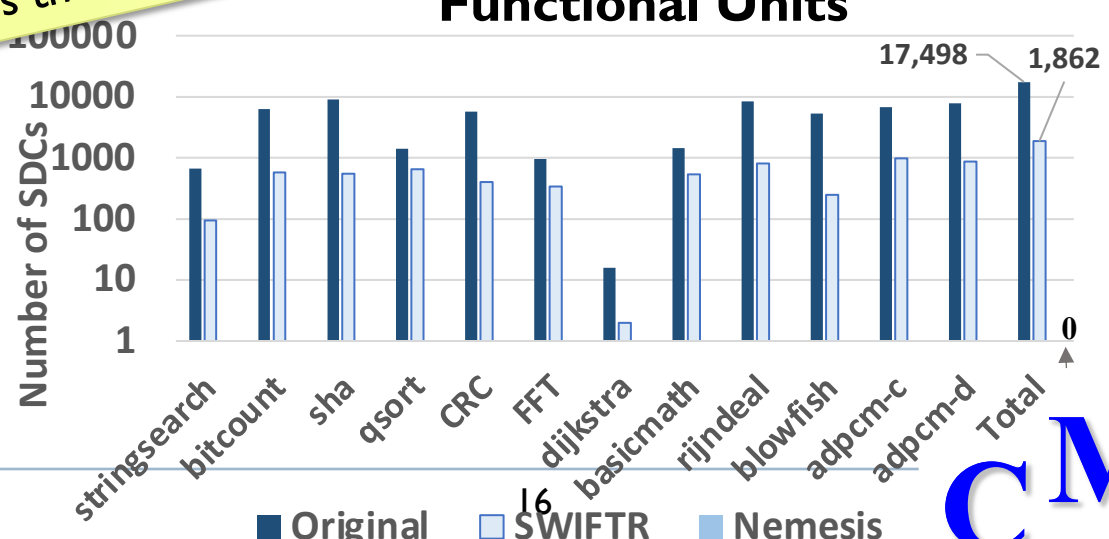
NEMESIS-protected programs rarely produce wrong result



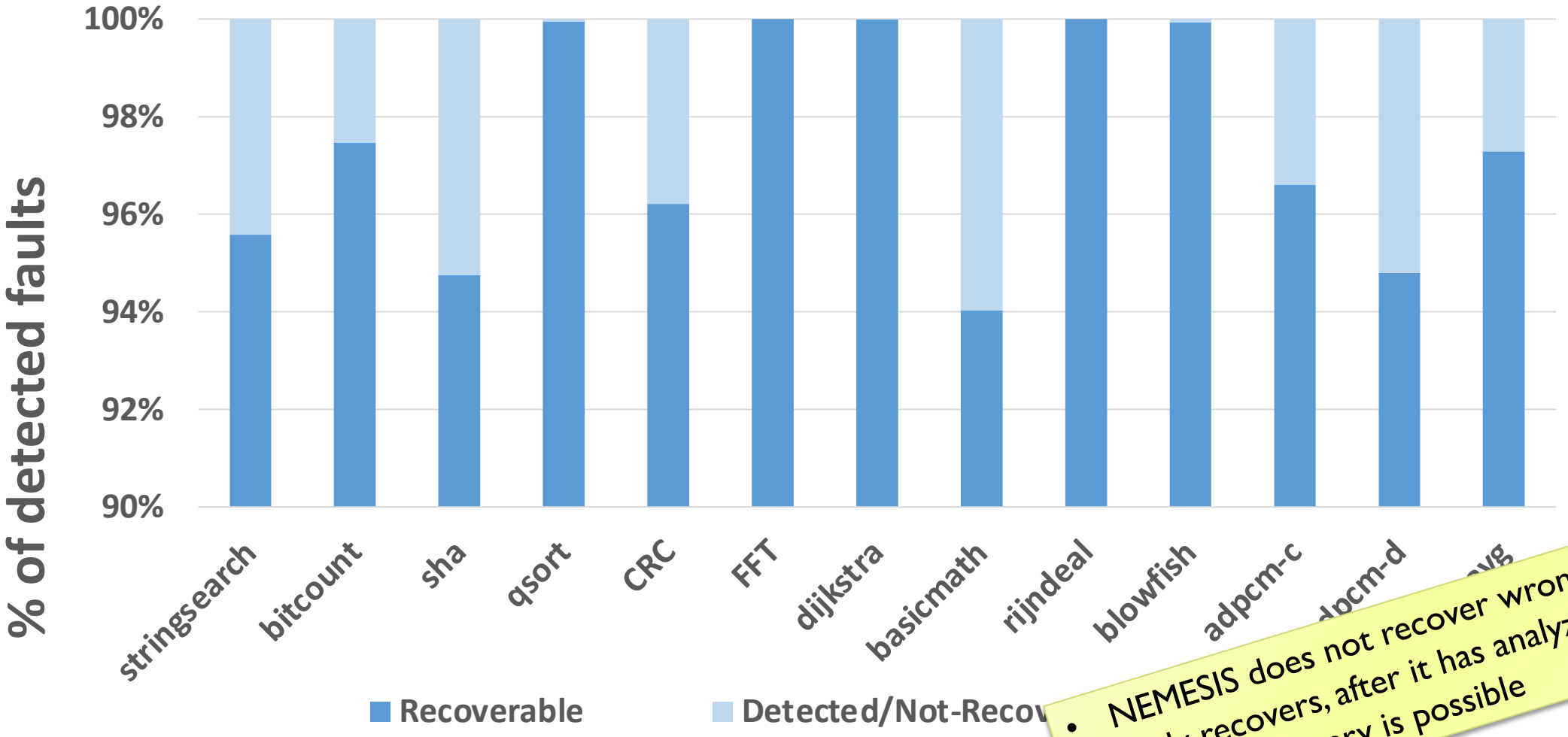


- Experiments performed with 95% confidence and 0.1% error rate
- NEMESIS provides 99.9% coverage (three 9s)
- SWIFTR provides ~ 90% coverage (one 9)
- Runtime 25% less than SWIFTR





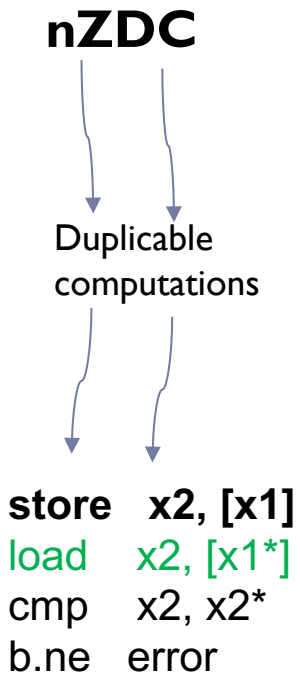
4% detected but un-recoverable errors

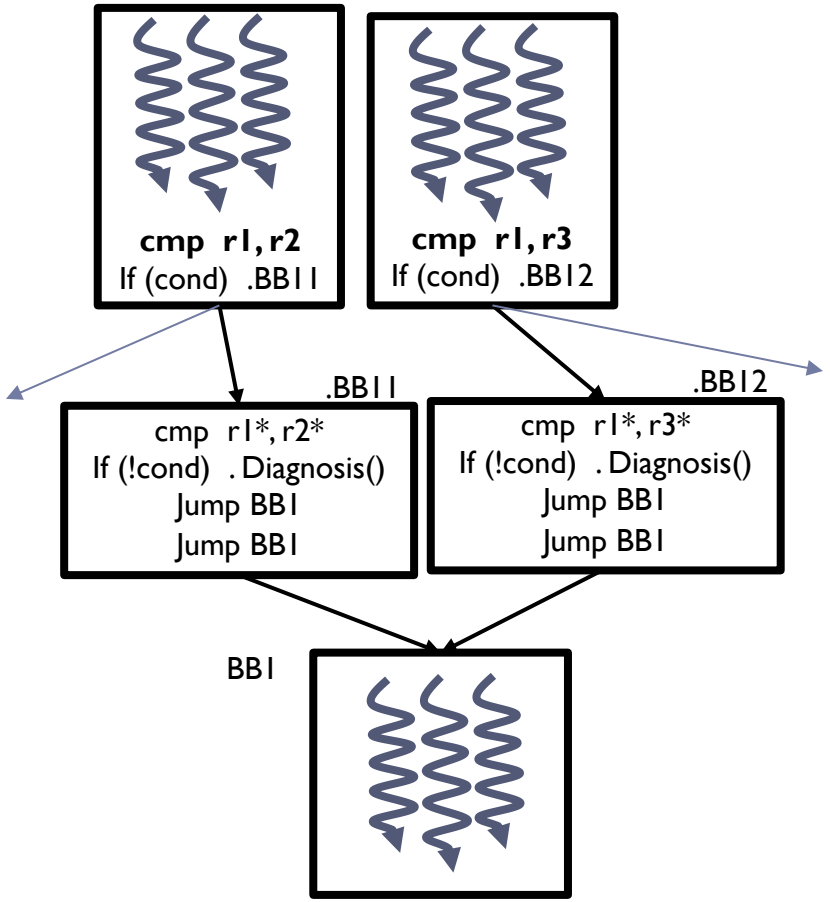


- NEMESIS does not recover wrong
- Only recovers, after it has analyzed that correct recovery is possible

Our approach - nZDC and Nemesis

- Main tenets of our approach
  - Lazy checking for error detection
  - Check to see if error is recoverable – do not recover wrong!!
  - Do not put recovery in the critical path
    - Only keep detection in the critical path
    - take time to thoroughly diagnose and recover





Summary

- Resilience in software is valuable – provides flexible protection
  - Error resilience only in the last (decision making) stages of ML inferencing
- In-time resilience is especially important for time-sensitive, safety-critical CPS
- Redundancy seems to be the only way to protect programs
  - But it is not as easy!!
- Existing software approaches do not achieve high coverage
  - Need fine-grain redundancy for in-time resilience
  - Pet peeve: A lot of resilience papers just evaluate the performance impact of the technique, and assume that arguments are enough to establish resilience
  - Extensive fault injection testing on RTL of our techniques show more than 99.9% (ASIL D compliant) coverage.
- Outlook – PL support for resilience
  - How to achieve do cross-layer protection is still a challenge
  - How to integrate with approximate computing is also an open challenge

Backup

LEAP-DICE

- **Positives**
  - Multi-bit error tolerant
  - In-time resilience

- **Negatives**
  - Only works for flip flops and registers, and not for combinational circuits.
  - Redesign of the processor
  - 2000X fewer soft errors in 180 nm design with 200 MeV proton radiation
  - 40% area cost
  - C-element/LEAP-dice can only provide detection. You still need a correction scheme. Having previous state only works for microarchitectural components, and not for the final commit.

LEAP-DICE

- LEAP (or Layout Design through Error-Aware Transistor Positioning), first introduced by Lilja [2008], is a new layout principle for soft error resilience of digital circuits [Lee et al., 2010]. According to the LEAP principle, given a circuit topology, without any modification of the circuit, it is possible to produce a soft-error-resilient layout, by performing:
- 1. An analysis of the circuit response to a single event for each individual drain contact node in the layout, and
- 2. A careful placement of each drain contact node in the layout based on the above analysis, such that multiple drain contact nodes act together to cancel (fully or partially) the overall effect of the single event on the circuit.

Unsolved problems

- Add becomes and add plus store – cannot be detected

Problems we solved

- Silent Store
  - Store is writing the same value to the memory, as there is in the memory
  - If error in the address, then readback from memory cannot detect error
    - First we check whether the store is silent, then we can just skip the store
    - Your test of silent store may fail… if the error was in address, and the other location had the same value.
- Wrong branch taken
  - Instead of checking the branch condition before the branch, execute the branch, and then re-execute from the shadow version to see, if the branch was right.

- C-element
  - Scan and test flops are there.
  - Those can be the redundant flip flops
  - C-element can detect mismatch between them
- Layout problem
  - Bist flip flops could be placed far from the required one. But in a C-element kind of design, they must be together, and that adds a latency constraint to the design.
- The C-element has a capacitor.
  - If the capacitor is hit, then it will cause an error
  - Keeper circuit
- 10X improvement in SER

EDDI proposed in 2002

Do not check for the address

There is 1 sp, and so, if there is 1 error, then there is a problem.

QED 2012 contains the load-back

but they have the wrong implementation of load back

EDDI with load back

Load back with EDDI does not make much sense, since if they are duplicating memory, then the differences will be there later also.

It makes sense for nZDC which has single memory, since there is only one opportunity to detect it.

EDDI vs nZDC

EDDI will reach to only 95-97% Need CFC. Need sp duplication.